VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SEBEMODIFIKUJÍCÍ SE PROGRAMY V KARTÉZSKÉM GENETICKÉM PROGRAMOVÁNÍ

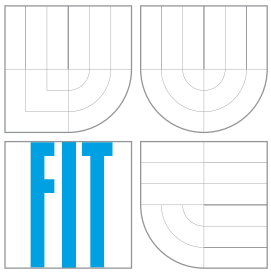
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MILOŠ MINAŘÍK

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SEBEMODIFIKUJÍCÍ SE PROGRAMY V KARTÉZSKÉM GENETICKÉM PROGRAMOVÁNÍ

SELF-MODIFYING PROGRAMS IN CARTESIAN GENETIC PROGRAMMING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MILOŠ MINAŘÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. LUKÁŠ SEKANINA, Ph.D.

BRNO 2010

Abstrakt

Kartézské genetické programování se během posledních let ukázalo jako velmi perspektivní oblast evolučních výpočtů. Má však jistá omezení, která znemožňují řešit pomocí něj rozsáhlejší nebo obecné problémy. Tato omezení lze eliminovat pomocí novějšího přístupu umožňujícího sebemodifikaci programů v kartézském genetickém programování. Cílem této práce je zhodnotit dosavadní vývoj a aktuální situaci v této oblasti a navrhnout vlastní řešení různých problémů, při jejichž řešení klasické kartézské genetické programování selhává. Jedním z těchto problémů, kterými se práce zabývá, je generování členů Taylorova rozvoje pro různé funkce. Vzhledem k tomu, že se jedná o problém vyžadující zobecnění, je cílem dokázat, že sebemodifikující varianta kartézského genetického programování je v tomto ohledu lepší než klasická. Dalším řešeným problémem bude využití sebemodifikujících programů v kartézském genetickém programování k návrhu řadicích sítí pro libovolný počet vstupů. Také v tomto případě je záměrem dokázat, že sebemodifikace přináší do kartézského genetického programování nové aspekty nutné k vývoji libovolně rozsáhlých řešení.

Abstract

During the last years cartesian genetic programming proved to be a very perspective area of the evolutionary computing. However it has its limitations, which make its use in area of large and generic problems impossible. These limitations can be eliminated using the recent method allowing self-modification of programs in cartesian genetic programming. The purpose of this thesis is to review the development in this area done so far. Next objective is to design own solutions for solving various problems that are hardly solvable using the ordinary cartesian genetic programming. One of the problems to be considered is generating the terms of various Taylor series. Due to the fact that the solution to this problem requires generalisation, the goal is to prove that the self-modifying cartesian genetic programming scores better than classic one for this problem. Another discussed problem is using the self-modifying genetic programming for developing arbitrarily large sorting networks. In this case, the objective is to prove that self-modification brings new features to the cartesian genetic programming allowing the development of arbitrarily sized designs.

Klíčová slova

evoluce, genetické programování, kartézské genetické programování, sebemodifikující kartézské genetické programování

Keywords

evolution, genetic programming, cartesian genetic programming, self-modifying cartesian genetic programming

Citace

Miloš Minařík: Sebemodifikující se programy v kartézském genetickém programování, diplomová práce, Brno, FIT VUT v Brně, 2010

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Doc. Ing. Lukáše Sekaniny, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Miloš Minařík
25. května 2010

Rád bych poděkoval svému vedoucímu práce Doc. Ing. Lukáši Sekaninovi, Ph.D. za cenné rady a připomínky, které mi v průběhu vytváření práce poskytnul. Dále bych chtěl poděkovat své přítelkyni a matce za pochopení, trpělivost a podporu při psaní této práce.

Obsah

Kapitola 1

Úvod

Evoluční techniky prodělaly během posledních let velký vývoj a dostaly se do popředí zájmu při hledání netradičních řešení různých problémů. Hlavní výhodou těchto technik je především to, že nejsou svázány zažitými postupy, kterých se při návrhu drží inženýři navrhující různé systémy. Díky tomu lze pomocí nich najít někdy i velmi překvapivá řešení.

Postupem času se evoluční techniky vyvíjely od jednoduchých k velmi komplexním, umožňujícím řešit i značně složité problémy. V poslední době vzrostl zájem o genetické programování, které oproti optimalizačním technikám hledajícím pouze nejvhodnější hodnoty parametrů předem daného návrhu umožňuje návrh přímo vytvářet.

Jednou z významných změn v oblasti genetického programování byla změna reprezentace jedince. Zpočátku byly programy reprezentovány pomocí stromů, v poslední době se však ukázala jako velmi vhodná reprezentace pomocí kartézského souřadného systému. Kromě toho se provádí více či méně úspěšné pokusy s dalšími typy reprezentace, například gramatikami.

Výše zmíněné kartézské genetické programování (CGP), se ukázalo jako velmi vhodné především pro návrh kombinačních logických obvodů. Má však jistá omezení, která neumožňují vyvíjet dostatečně obecná řešení. Tato omezení lze obejít pomocí sebemodifikujících programů, jimiž se v této práci budu zabývat. Základní informace o problematice CGP jsem čerpal z různých zdrojů, například [12] či [14]. Vzhledem k tom, že sebemodifikující kartézské genetické programování (SMCGP) je poměrně nový přístup, čerpal jsem informace o něm především z článků z různých konferencí (např. [3]).

Cílem této diplomové práce je popsat problematiku kartézského genetického programování a jeho sebemodifikující varianty. Kromě vysvětlení základních pojmů a principů funkce je zde ve formě výsledků různých experimentů shrnutý současný stav v této oblasti. Dále následuje popis mé vlastní implementace SMCGP a několik na ní provedených experimentů.

Druhá kapitola této práce se zabývá koncepcí kartézského genetického programování, která je nutná k pochopení problematiky řešené v rámci této práce. Je zde uveden způsob zakódování jedinců, postup evoluce, použitelné genetické operátory a další.

Třetí kapitola je věnována sebemodifikujícímu kartézskému genetickému programování (SMCGP). Popisuje změny provedené oproti klasickému kartézskému genetickému programování a jejich vliv na tento způsob genetického programování jako takový.

Ve čtvrté kapitole jsou uvedeny aplikace SMCGP. Jedná se o shrnutí několika publikovaných experimentů, které budou v pozdější práci využity jako základ pro návrh vlastních experimentů a srovnání výsledků. Tyto experimenty nejsou rozebírány nijak podrobně, většinou je uveden pouze cíl experimentu, nastíněn postup a vypsány dosažené výsledky.

Pátá kapitola obsahuje popis rozšíření, která jsem zavedl oproti konceptu SMCGP z [4].

Tato rozšíření by měla zlepšovat původní SMCGP ať již po stránce obecnosti nebo rychlosti vyhodnocení.

V šesté kapitole jsou popsány základy konkrétní implementace. Jsou zde obsaženy základní datové typy, konstanty, struktury a funkce potřebné pro pochopení této konkrétní implementace, pokud by chtěl čtenář program nějakým způsobem upravovat. Jedna část je pak věnována problémům, na které jsem při implementaci narazil.

Sedmá kapitola obsahuje několik částí, přičemž v každé části je popsán jeden experiment. Jednotlivé části obsahují vždy popis problému, návrh experimentu a dosažené výsledky. Navíc jsou vždy uvedeny konkrétní hodnoty důležitých proměnných určujících chování systému. Je tomu tak proto, aby bylo možné experimenty reprodukovat ve stejných podmínkách.

Poslední kapitola obsahuje shrnutí dosažených výsledků a možná další rozšíření, která jsem v rámci této práce neimplementoval.

Kapitola 2

Kartézské genetické programování

2.1 Historie

Původně se pomocí genetického programování (GP) vyvíjely programy ve formě stromů, popř. výrazů jazyka LISP [9], přičemž se nerozlišoval genotyp a fenotyp. Stromy jsou speciálním případem grafů, ve kterých může být mezi dvěma uzly maximálně jedna cesta. Používaly se většinou velké populace a jako primární metoda vývoje nových kandidátních řešení ze starších programů se využívalo křížení. To byl jeden z původních důvodů pro používání stromů, protože tak lze použít křížení na genotypy různé délky.

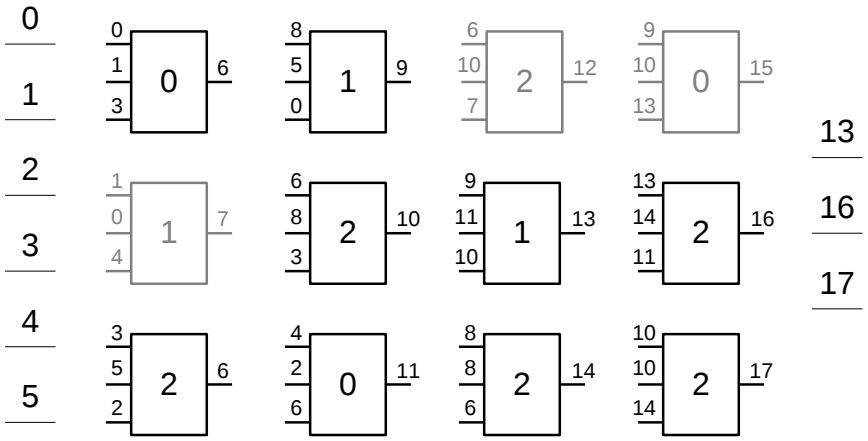
Dalším odvětvím evolučních technik bylo evoluční programování (EP). Na rozdíl od genetického programování se v EP uplatňovala spíše mutace. Evoluční techniky se používaly i pro návrh číslicových obvodů. Právě v této oblasti výzkumu se začaly používat samostatné funkční jednotky implementující různé funkce (AND, OR, sčítačky), přičemž se evolučně vyvíjelo jejich vzájemné propojení. Nejprve se k tomu využívaly evoluční algoritmy, které pracovaly s řetězci celých čísel popisujícími propojení komponent, například [13]. Od tohoto přístupu již nebylo daleko ke kartézskému genetickému programování. To bylo představeno v roce 2000 [12]. Kartézský způsob reprezentace vznikl především proto, že přibližně odpovídá uspořádání FPGA.

2.2 Reprezentace programu

V CGP se k reprezentaci programu používají orientované grafy, které jsou obecnější než stromy. Takový graf lze zakódovat jako řetězec celých čísel, který kóduje propojení uzlů grafu a jejich funkce.

V CGP jsou uzly uspořádány do dvourozměrné mřížky. Odtud ostatně pochází i název tohoto přístupu. Jak již bylo zmíněno, je tento způsob reprezentace velmi vhodný pro návrh obvodů, např. v FPGA.

Mřížka má pro daný problém stále pevnou velikost. Počet řádků mřížky označíme n_r, počet sloupců n_c. Počet vstupů obvodu označíme n_i, počet výstupů n_o. Každému uzlu je přiřazena funkce z množiny F obsahující n_f prvků. Počet vstupů každého uzlu označíme n_a. Posledním významným parametrem je hodnota l. Tato hodnota určuje, kolik předchozích sloupců může své hodnoty připojovat ke vstupům uzlů v aktuálním sloupci. Pokud je tedy parametr l nastaven na hodnotu 1, lze na vstupy připojovat pouze výstupy uzlů v předchozím sloupci. Naproti tomu, pokud určíme $l = n_c$, lze uzly propojovat libovolně. Když současně platí $n_r = 1$, je umožněna nejvyšší možná konektivita. Nastavení tohoto



Obrázek 2.1: Zakódování programu v CGP

parametru tedy výrazně ovlivňuje velikost stavového prostoru.

Význam jednotlivých hodnot si vysvětlíme na jednoduchém příkladu. Uvažme mřížku o 3 řádcích a 4 sloupcích, tedy $n_r = 3$, $n_c = 4$. Uzly budou mít 3 vstupy a množina funkcí F, které lze uzlům přiřadit, bude obsahovat 3 prvky, tedy $n_a = 3$, $n_f = 3$. Dále uvažujme, že při řešení problému budeme využívat 5 primárních vstupů a 3 primární výstupy, tedy $n_i = 5$, $n_o = 3$. Vzorový genotyp pro takto definovaný problém tedy může vypadat například takto:

(0 1 3 0), *(1 0 4 1)*, (3 5 2 2), (8 5 0 1), (6 8 3 2), (4 2 6 0), *(6 10 7 2)*, (9 11 10 1), (8 8 6 2), *(9 10 13 0)*, (13 14 11 2), (10 10 14 2), (13 16 17)

Na obrázku 2.1 je znázorněn fenotyp, který přísluší uvedenému genotypu. Převod genotypu na fenotyp probíhá tak, že se začne u výstupů a postupuje se zpět k uzlům nutným pro určení hodnoty těchto výstupů. Všimněme si, že ve zobrazeném fenotypu je tedy aktivních (potřebných pro určení výstupů) pouze 9 uzlů.

Jak je z výše uvedených informací patrné, mají v CGP genotypy pevnou délku. Tato délka je rovna $n_r n_c(n_a + 1) + n_o$. Oproti tomu fenotypy nabývají různé délky podle počtu aktivních genů. Délka fenotypu je tedy omezena pouze shora, a to délkou genotypu. Důležitost mapování genotypu na fenotyp byla prokázána v binárním genetickém programování (BGP) [1]. U BGP se binární řetězce přeloží a opraví tak, aby tvořily platné programy. V CGP není žádná oprava třeba. Další výhodou využití mapování genotypu na fenotyp je možnost, aby se několik genotypů mapovalo na stejný fenotyp. Tato vlastnost zajišťuje neutralitu, která bude popsána později.

2.3 Redundance

V CGP se vyskytuje velký počet genotypů, které se mapují na stejné fenotypy. Je tomu tak kvůli značné redundanci.

Jedná se například o redundanci uzlů, která je způsobena geny uzlů, jenž nejsou součástí souvislého grafu znázorňujícího program. Tato redundance se může během evoluce měnit.

Dalším typem redundance v CGP je funkční redundance, která se však vyskytuje i u ostatních druhů GP. V tomto případě je určitá funkce implementována určitým počtem

uzlů, který je vyšší, než počet uzlů nutný k její implementaci. Toto zvyšování počtu redundantních uzlů tvoří tzv. bloat (viz [10]). Tento jev je v GP poměrně častý, avšak nežádaný.

Třetím typem je redundance vstupů. Tato redundance vzniká v případě, že funkce uzlu nevyužívá některé vstupy. Příkladem takové redundance může být například uzel realizující funkci NOT, který má 3 vstupy.

Redundance uzlů a vstupů může být užitečná ve smyslu zvýšení neutrality (viz dále). Například uzel, který není v aktuálním genotypu aktivní, může projít neutrální změnou a být připojen později. Tato možnost může v mnoha případech způsobit dosažení vyšší hodnoty fitness. Uzel s redundantními vstupy může například projít mutací, která přiřadí uzlu funkci s jinou aritou. Příkladem může být opět funkce NOT, která se mutací změní na funkci AND. V tom případě se dříve redundantní uzel stává rázem užitečným. Funkční redundance pravděpodobně pozitivně přispívá k prohledávání, protože zvyšuje počet možných způsobů, kterými lze implementovat určitou funkci. Protože se v CGP mohou uzly i odpojovat, nevyskytuje se tzv. bloat tak často jako u jiných přístupů ke GP [11].

2.4 Neutralita

Neutralita znamená, že se mohou vyskytnout různé genotypy se stejnou hodnotou fitness. Předpokládá se, že při umělé evoluci může neutralita zajistit nalezení cesty, díky níž může evoluce překonat oblasti s nízkým ohodnocením fitness. Tento názor podporuje například studie mapování genotypu na fenotyp při řešení optimalizačního problému s omezujícími podmínkami pomocí genetického programování (viz [1]). Je vhodné zmínit, že neutralita způsobená redundancí genotypů je vhodná pouze v případě, že neutrální změny v budoucnu ovlivní genotyp. Pouhé přidání neaktivních genů neutrální evoluci neumožní.

2.5 Evoluční operátory a strategie

Moderní GP umožňuje používat i automaticky definované funkce (ADF). V CGP automaticky definované funkce neexistují. Nahrazuje je zde možnost opakovaného použití výstupů. Výhodou tohoto přístupu je to, že k opakovanému použití výstupu není třeba žádný zvláštní způsob zakódování genotypu. Tento přístup má však i jisté nevýhody. Výstupy lze sice použít vícekrát, avšak bude se vždy jednat o výstupy založené na stejných vstupech. To je důvod, proč není opakované použití výstupů natolik obecné jako ADF. Opakované použití výstupů lze ovlivnit i určením parametrů mřížky. Pokud bude mřížka tvořena jedním řádkem,bude šance na opakované použití určitého výstupu nejvyšší. Naopak, pokud by mřížka měla pouze jeden sloupec, nebylo by výstupy možné znovu použít.

Jako evoluční operátor se v CGP používá mutace, i když při dodržení určitých podmínek je možné použít i křížení (viz [12]). Při provádění mutací je třeba dávat pozor na to, aby zůstala dodržena daná omezení (především parametr l). Samotná mutace může měnit funkci uzlu a propojení vstupů a výstupů s ostatními uzly. Mutace je velmi důležitá především proto, že může z neaktivních uzlů učinit aktivní a naopak. Pokud se po provedení mutace nezmění hodnota fitness, mohou existovat dva důvody:

- Fenotyp se změnil na jiný fenotyp se stejnou hodnotou.
- Mutace ovlivnila uzly genotypu, které nejsou ve fenotypu aktivní. V takovém případě se jedná o neutrální mutaci.

V CGP se využívá obdoba evoluční strategie $1 + \lambda$. Tato strategie spočívá v tom, že generaci tvoří $1 + \lambda$ jedinců. Z této generace se vybere jedinec s nejlepší hodnotou fitness a vloží se do následující generace. Zbytek nové generace se doplní jedinci, kteří vzniknou mutací tohoto jedince. Parametr λ má většinou hodnotu 4.

Algoritmus evoluce je následující:

1. Náhodné vygenerování počáteční populace
2. Vyhodnocení hodnoty fitness genotypů v populaci
3. Převod nejlepšího genotypu do nové populace
4. Vyplnění zbývajících míst v populaci mutacemi nejlepšího jedince
5. Zpět na krok 2, dokud není splněno požadované kritérium.

Důležitou změnou v CGP je to, že pokud existuje více nejlepších genotypů, které mají stejnou hodnotu fitness, potom se jako nový rodič vybere náhodně některý z těchto jedinců. Tomuto postupu se říká neutrální prohledávání. Pokud se vybírají pouze jedinci, kteří mají hodnotu fitness vyšší, tak se o neutrální prohledávání nejedná.

Kapitola 3

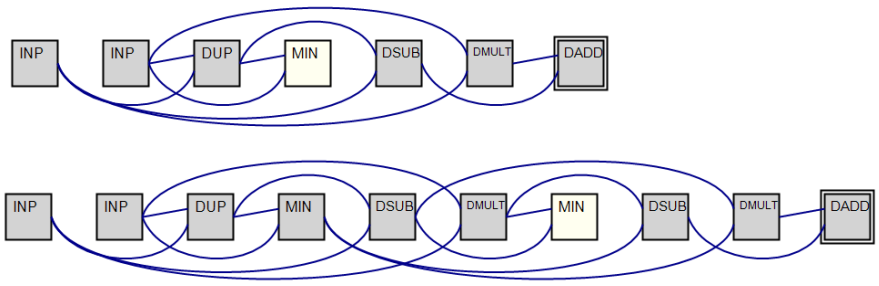
Sebemodifikující kartézské genetické programování

3.1 Motivace

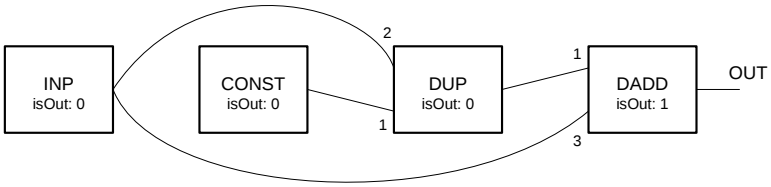
V biologii lze za sebemodifikaci považovat proces, při kterém se z genotypu stává fenotyp. V průběhu dekódování je vyjádření genu ovlivněno celou řadou faktorů okolního prostředí [7]. Na sebemodifikaci lze tedy pohlížet jako na vývoj umožňující zahrnutí procesů genetické regulace v jednotlivých buňkách, přepisování grafů a mnohobuněčných systémů.

V evolučních výpočtech se dříve sebemodifikace příliš nevyužívala. Poté však několik experimentů prokázalo její nesporný přínos při tvorbě vyvíjejících se programů (např. [15]). Hlavní význam vyvíjejících se systémů tkví především v tom, že tímto způsobem lze evolučně vyvíjet libovolně velké systémy, které by pomocí přímé genetické reprezentace nebylo možné vyvinout.

Za sebemodifikaci označujeme proces, jehož pomocí se fenotyp může měnit i po ukončení evoluce. To umožňují speciální funkce v množině funkcí uzlů F. Tyto funkce mohou manipulovat s uzly fenotypu (kopírovat je, odstraňovat, přesouvat, měnit propojení atd.). Příklad takové sebemodifikace je znázorněn na obrázku 3.1, kde se pomocí funkce DUP duplikují uzly, takže fenotyp má v následující iteraci větší délku.



Obrázek 3.1: Sebemodifikace programu v SMCGP: Duplikace uzlů pomocí funkce DUP podle [6]



Obrázek 3.2: Zakódování chromozomu v SMCGP

3.2 Zakódování

Zde uvedené zakódování je převzato z [3]. Toto zakódování není samozřejmě závazné a lze jej upravovat. Bude však použito jako příklad pro vysvětlení základních principů využívaných v SMCGP. Vzhledem k tomu, že při nastavení mřížky na jeden řádek a n_c sloupců a předpokladu $L = n_c$ jsou možnosti propojení nejlepší, budeme dále uvažovat pouze tuto možnost. Stejně jako v CGP i zde určuje první celé číslo v zakódování uzlu funkci uzlu. Následující celá čísla určují uzly grafu, jejichž výstupy se berou jako vstupy tohoto uzlu. Oproti CGP však SMCGP obsahuje i tři reálné hodnoty, které zastupují parametry potřebné pro funkci uzlu. Konkrétní význam těchto parametrů je popsán dále v části věnované sebemodifikujícím funkcím. Následující hodnota určuje, zda se má výstup tohoto uzlu použít jako výstup programu. Příklad zakódování chromozomu je na obrázku 3.2.

Stejně jako u CGP používají uzly jako vstup výstupy z předchozích uzlů nebo vstupy programu (primární vstupy). Na rozdíl od CGP však nejsou pozice vstupů vyjádřeny absolutně, ale relativně vzhledem k pozici aktuálního uzlu. Tomuto relativnímu vyjádření pozice vstupu budeme říkat adresa. Hodnota 2 tedy například znamená, že se jako vstup použije výstup uzlu, který je o dva sloupce zpět. Aby byla zajištěna acykličnost výsledného grafu, je nutné, aby byly hodnoty určující vstupy větší než 0.

Pokud se gen odkazuje na vstup ležící mimo graf (například, když je relativní adresa větší než počet uzlů, ke kterým se lze připojit), předpokládá se, že je na tomto vstupu hodnota 0. Oproti klasickému CGP se vstupy programu zavádějí pomocí speciálních funkcí. Relativní adresování spojení umožňuje přemístit nebo duplikovat podgrafy tak, že si zachovají sémantickou platnost.

Tři geny parametrů funkce uzlu se používají především při provádění modifikací fenotypu. V genotypu jsou reprezentovány reálnými čísly, ale pro potřeby určitých funkcí je lze zaokrouhlit. Dostupné funkce a jejich parametry jsou popsány v části věnované funkcím uzlů.

3.3 Vstupy a výstupy

Způsob práce se vstupy popsaný v [3] (dále jako původní postup) se ukázal jako nevhodný. Sebemodifikující funkce při přesouvání podgrafů dále od počátku grafu posunovaly vstupy takovým způsobem, že nebyla zachována sémantická platnost. V článku [4] (dále jako nový postup) tedy byly provedeny následující změny.

V původním postupu se záporné adresy mapovaly na vstupy programu. V novém postupu se vstupy se zápornou adresou nastavují na hodnotu 0.

Druhou změnou je odlišná funkce INPUT. Když má uzel přiřazenou funkci INPUT,

vezme se při každém volání této funkce další hodnota z dostupné množiny vstupů. Pokud je funkce INPUT volána vícekrát, než je počet vstupních položek, začne počítání od začátku a použije se první uzel.

Výstupy se zpracovávají jiným způsobem. Jak je uvedeno výše, obsahuje každý uzel binární hodnotu určující, zda bude výstup daného uzlu výstupem programu. V původním postupu se jako výstupy programu používaly výstupy posledních n_o uzlů grafu. Stejně jako u vstupů se však tento postup neosvědčil, protože při růstu velikosti grafu nebyly výstupem žádoucí hodnoty. V novém postupu se tedy při vyhodnocování jedince nejprve určí všechny uzly grafu, jejichž výstupní gen je nastaven na hodnotu 1. Jakmile jsou tyto uzly nalezeny, provede se rekurzivně jejich vyhodnocení.

V případě, že nejsou jako výstupní označeny žádné uzly, použije se posledních n_o uzlů grafu. V tom případě jsou výstupy stejné jako u původního postupu. Když je jako výstup označeno více uzlů, než je výstupů programu, použije se n_o nejlevějších uzlů. Jestliže je výstupních uzlů méně než výstupů programu, považuje se jedinec za poškozeného a jeho hodnota fitness se příslušným způsobem upraví tak, aby se nedostal do následující generace.

3.4 Množina funkcí

Množina funkcí se skládá ze dvou částí. První část obsahuje operátory pro sebemodifikaci. Ty jsou společné pro všechny datové typy používané v SMCGP. Tyto funkce jsou vybrány tak, aby pokrývaly co nejvíce operací modifikace. Druhou částí množiny jsou výpočetní operace, jejichž zpracování je dáno konkrétním řešeným problémem.

Způsob vyhodnocení sebemodifikujících funkcí je určen 4 proměnnými. První tři proměnné již byly zmíněny v části věnované zakódování. Nazývají se parametry a značí se P_0, P_1, P_2. Čtvrtou proměnnou je pozice uzlu obsahujícího sebemodifikující funkci v rámci grafu fenotypu. Tato proměnná se značí x a její hodnota leží v rozsahu 0 až celkový počet uzlů grafu, přičemž hodnotě 0 odpovídá nejlevější uzel. V definicích funkcí se často využívají geny propojení. Gen j-tého spojení uzlu na pozici i se značí c_{ij}.

Zde je několik pravidel, která určují, jak se zachází s adresami a parametry:

- Výsledkem součtu parametrů P_i a x je celé číslo.
- Pokud jsou adresy mimo rozsah grafu, upraví se. Adresy menší než 0 se považují za nulové. Adresy, které se nacházejí za hranicí grafu, se oseknou na délku grafu.
- Počáteční a koncové indexy se seřadí ve vzestupném pořadí (je-li to nutné).
- Redundantní operace (např. kopírování 0 uzlů) se ignorují, započítávají se však do celkové délky seznamu "To Do".

3.5 Vyhodnocení grafu SMCGP

Vyhodnocení genotypu probíhá v zásadě tak, že se z genotypu vytvoří fenotyp. Tento graf se následně vyhodnotí a v případě, že jsou vyhodnocovány uzly se sebemodifikujícími funkcemi, provedou se a změní graf fenotypu.

Genotyp zůstává po celou dobu vyhodnocení beze změny. Všechny modifikace se provádí na fenotypu. Během iterací se fenotyp většinou čím dál více odlišuje od původního genotypu.

Tabulka 3.1: Sebemodifikující funkce

Název funkce	**Popis**
Duplikace a škálování adres (DU4)	Zkopíruje P_1 uzlů od pozice $P_0 + x$ a vloží je za uzel na pozici $P_0 + x + P_1$. Během kopírování se hodnoty c_{ij} kopírovaných uzlů vynásobí hodnotou P_2.
Posun propojení (SHIFTCONNECTION)	U P_1 uzlů od indexu $P_0 + x$ přičte k hodnotám c_{ij} parametr P_2.
Posun propojení násobením (MULTCONNECTION)	U P_1 uzlů od indexu $P_0 + x$ vynásobí hodnoty c_{ij} parametrem P_2.
Přesun (MOV)	Přesune uzly mezi pozicemi $P_0 + x$ a $P_0 + x + P_1$ a vloží je za uzel na pozici $P_0 + x + P_2$.
Duplikace (DUP)	Zkopíruje uzly mezi pozicemi $P_0 + x$ a $P_0 + x + P_1$ a vloží je za uzel na pozici $P_0 + x + P_2$.
Duplikace se zachování propojení (DU3)	Zkopíruje uzly mezi pozicemi $P_0 + x$ a $P_0 + x + P_1$ a vloží je za uzel na pozici P_0+x+P_2. Během kopírování mění tato funkce hodnoty c_{ij} kopírovaných uzlů tak, že se stále odkazují na původní uzly.
Odstranění (DEL)	Odstraní uzly mezi pozicemi $P_0 + x$ a $P_0 + x + P_1$.
Přidání (ADD)	Přidá P_1 nových uzlů za uzel na pozici $P_0 + x$.
Změna funkce (CHF)	Změní funkci uzlu P_0 na funkci spojenou s P_1.
Změna propojení (CHC)	Změní $(P_1 mod 3)$-té propojení uzlu P_0 na P_2.
Změna parametru (CHP)	Změní $(P_1 mod 3)$-tý parametr uzlu P_0 na $P2$.
Přepsání (OVR)	Zkopíruje uzly mezi pozicemi $P_0 + x$ a $P_0 + x + P_1$ na pozici $P_0 + x + P_2$, přičemž nahradí existující uzly na cílové pozici.
Kopírovat po zastavení (COPYTOSTOP)	Zkopíruje uzly od x po další uzel s funkcí COPYTOSTOP, uzel STOP nebo konec grafu. Uzly se vloží na pozici, na které se operace zastaví.

Tabulka 3.2: Numerické a další funkce

Název funkce	**Popis**
Prázdná operace (NOP)	Předá první vstup.
Součet, rozdíl, součin, podíl (DADD, DSUB, DMULT, DDIV)	Provede se dvěma vstupy příslušnou matematickou operaci.
Konstanta (CONST)	Vrátí číselnou konstantu určenou parametrem P_0.
$\sqrt{x}$, $\frac{1}{\sqrt{x}}$, Cos, Sin, TanH, absolutní hodnota (SQRT, DRCP, COS, SIN, TANH, DABS)	Použije na první vstup příslušnou operaci (druhý vstup se ignoruje).
Průměr (AVG)	Vrátí aritmetický průměr dvou vstupů.
Index uzlu (INDX)	Vrátí index aktuálního uzlu, přičemž první uzel má index 0.
Počet vstupů (INCOUNT)	Vrátí počet vstupů programu.
Min, Max (MIN, MAX)	Vrátí minimum nebo maximum ze dvou vstupů.

Graf se vyhodnocuje stejně jako u CGP s tím rozdílem, že je nutné zpracovávat navíc i sebemodifikaci. Vyhodnocování grafu začíná u výstupních uzlů a postupuje rekurzivně až ke vstupům programu. Tento přístup je výhodný především z toho důvodu, že se nevyhodnocují neaktivní uzly, což přispívá k urychlení vyhodnocení. To je velmi zásadní, protože vyhodnocování jedinců je časově nejnáročnější.

Běžné funkce, které neprovádí sebemodifikaci, se vyhodnotí stejným způsobem jako v běžném CGP. Výstup uzlu se tedy získá tak, že se na vstupní hodnoty použije funkce uzlu.

U sebemodifikujících funkcí je postup složitější. Na začátku se porovnají dvě vstupní hodnoty. Pokud je větší druhá hodnota, neprovede se žádná modifikace a tato hodnota projde tímto uzlem dále. Jestliže je však větší první hodnota, přidá se sebemodifikující funkce do seznamu "To Do". I v tomto případě projde tato hodnota uzlem dále. Hlavním přínosem tohoto přístupu je, že provedení modifikace závisí na vstupních hodnotách uzlu, takže chování grafu může být pro různé sekvence vstupních hodnot různé.

Po každé iteraci se projde seznam "To Do"a provedou se v něm uložené modifikace. Tento seznam je realizován jako fronta typu FIFO, nejprve jsou tedy provedeny modifikace ležící v grafu blíže k výstupním uzlům.

Vzhledem k tomu, že jsou modifikace poměrně výpočetně náročné, je možné omezit maximální velikost seznamu "To Do". Je však třeba si uvědomit, že přílišné omezení velikosti tohoto seznamu může výrazně degradovat přínos sebemodifikací.

3.6 Fitness funkce

Hlavní výhodou SMCGP je možnost hledání obecných řešení. Z toho však plyne problém, jak taková řešení ohodnotit pomocí fitness funkce. Ve většině případů není možné ověřit všechny výstupy. To může vyplývat již z podstaty řešeného problému. Pokud má program například hledat členy nekonečné řady, není technicky možné všechny tyto členy zkontrolovat. I když je řešení konečný počet, může jejich vyhodnocení trvat poměrně dlouho.

Z těchto důvodů se používají fitness funkce, které testují pouze několik prvních výstupů programu. V případě, že jsou všechny výstupy správné, ověří se obecnost řešení. To lze provést ověřením dalších výstupů až do požadovaného počtu nebo například pomocí nějaké analytické metody, pokud je to možné.

Vzhledem k tomu, že je počet výstupů testovaných ve fitness funkci omezen, mohou se jako řešení označit i jedinci, kteří generují správně výstupy kontrolované ve fitness funkci, ale další výstupy jsou již rozdílné od požadovaných. Taková řešení jsou však vyloučena během testu obecnosti.

Kapitola 4

Příklady využití SMCGP

Cílem této části je popsat současný stav aplikací SMCGP. K tomu poslouží příklady problémů úspěšně řešených pomocí SMCGP. Tyto experimenty zobrazují možnosti využití SMCGP a zároveň poskytují srovnání s ostatními metodami . Rovněž slouží jako základ k úvahám o možných vylepšeních rozebraných v dalších kapitolách této práce.

4.1 Druhé mocniny

Cílem tohoto experimentu bylo vyvinout pomocí SMCGP program, který vypisuje posloupnost druhých mocnin přirozených čísel (0, 1, 4, 9, 16, ...) bez použití operací násobení a dělení [4]. Jak bylo dokázáno v článku [15], musí být program pro úspěšné vyřešení této úlohy schopen modifikovat sám sebe. Důvodem je to, že k vyčíslení následujícího čísla v posloupnosti do sebe musí program přidat novou funkčnost.

Pro řešení tohoto problému byly použity všechny sebemodifikující funkce a numerické funkce DADD, CONST, INP, MIN, MAX, INDX, SORT a INCOUNT.

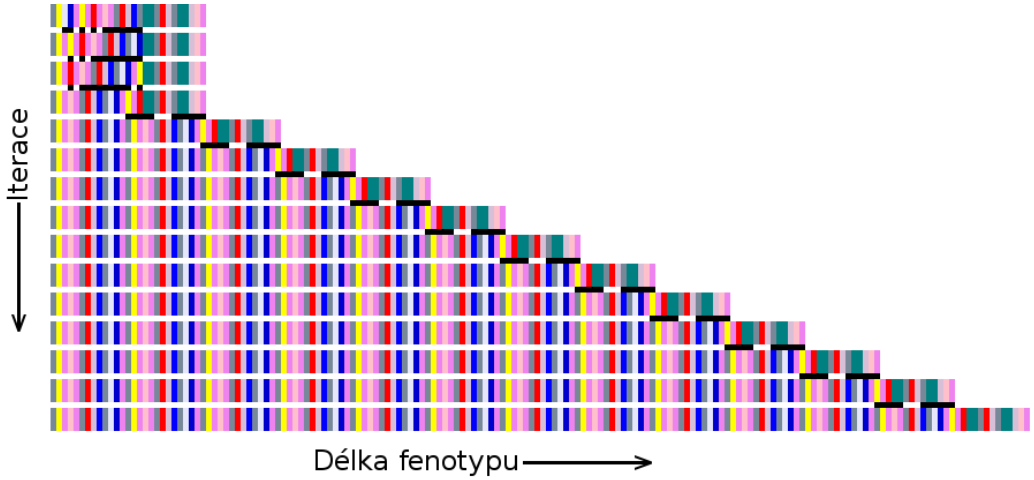
Programy byly vyvíjeny tak, aby dokázaly určit prvních 10 čísel posloupnosti, přičemž jako hodnota fitness byl využit počet správně vypsaných čísel. Poté, co byl úspěšně vyvinut program vypisující správnou posloupnost, byl testován, zda dokáže zobecňovat. Tento test zahrnoval výpis prvních 100 čísel posloupnosti. Bylo zjištěno, že 84.3% nalezených řešení bylo obecných. Shrnutí výsledků je uvedeno v tabulce 4.1.

Zajímavým jevem je lineární zvětšování délky fenotypu v průběhu iterací. Tento vývoj je jasně patrný na obrázku 4.1. Počet aktivních uzlů v grafu na počátku kolísal, ale přibližně od 15. generace se ustálil.

Dalším podobným experimentem bylo například generování Fibonacciho posloupnosti [4]. I v tomto experimentu se SMCGP ukázalo jako vhodná technika pro hledání obecného řešení.

Tabulka 4.1: Shrnutí výsledků hledání druhých mocnin podle [4]

Prům. poč. vyhodnocení	Směr. odch.	Min. poč. vyh.	Max. poč. vyh.	% obecných
141 846	513 008	392	3 358 477	84,3



Obrázek 4.1: Zvětšování délky fenotypu během iterací podle [4]

Tabulka 4.2: Počet evolucí potřebný pro vývoj programu sčítajícího posloupnost čísel dané délky podle [4]

Velikost množiny	Průměr	Minimum	Maximum	Směr. odch.	% CGP
2	50	50	50	0	100
3	618	54	3 248	566	80
4	1 266	64	9 334	1 185	95,8
5	1 957	116	9 935	1 699	48
6	2 564	120	11 252	2 151	38,1
7	3 399	130	17 798	2 827	0
8	4 177	184	17 908	3 208	0
9	5 138	190	18 276	3 871	0
10	5 918	201	22 204	4 401	0

4.2 Součet čísel

Cílem tohoto experimentu bylo vytvořit program, který dokáže sečíst libovolně dlouhý seznam čísel [4]. Po provedení n-té iterace by měl být program schopen sečíst seznam čísel o délce n. Tento problém byl zvolen kvůli své předpokládané obtížnosti pro genetické programování. Tento předpoklad byl vyvozen z úvahy, že pro kombinování vstupních hodnot jsou vhodné spíše jiné techniky (např. neuronové sítě), zatímco GP je vhodné spíše pro výběr prvků ze vstupu.

Vstupní vektory se skládaly z náhodných posloupností celých čísel. Jako hodnota fitness se brala celková absolutní chyba výstupu programu oproti předpokládanému výstupu. Vývoj programů probíhal s posloupnostmi o délce 2 až 10 čísel. Z funkcí uzlů byly vybrány všechny sebemodifikující funkce a z numerických funkcí pouze funkce ADD. Počet vyhodnocení nutný k dosažení n-tého součtu je uveden v tabulce 4.2.

Po skončení evoluce byla u nejlepších jedinců z jednotlivých generací testována obecnost řešení. Test spočíval v sečtení posloupnosti 100 čísel. Ze všech těchto řešení nebylo obecných pouze 0,07%.

V rámci tohoto experimentu byly provedeny pokusy o nalezení takového programu po-

Tabulka 4.3: Shrnutí výsledků klasifikace v oblasti bioinformatiky podle [4]

-	**CGP**	**SMCGP**
Průměrná hodnota fitness (trénování)	66,81	66,83
Směrod. odch. hodnoty fitness (trénování)	6,35	6,45
Průměrná hodnota fitness (ověření)	66,10	66,18
Směrod. odch. hodnoty fitness (ověření)	5,96	6,46
Prům. počet vyhodnocení pro nejlepší fitness (trénování)	7 679	7 071
Směrod. odch. počtu vyhodnocení pro nejlepší fitness (trénování)	2 452	2 644
Prům. počet vyhodnocení pro nejlepší fitness (ověření)	7 357	7 161
Směrod. odch. počtu vyhodnocení pro nejlepší fitness (ověření)	2 386	2 597

mocí klasického CGP. Výsledky těchto pokusů jsou uvedeny v posledním sloupci tabulky 4.2. Jak je z této tabulky patrné, klasické CGP dokázalo nalézt řešení pouze ve zlomku případů nebo vůbec. Toto porovnání zcela jasně ukazuje výhody sebemodifikujícího se kartézského genetického programování oproti CGP.

4.3 Klasifikace v bioinformatice

Cílem tohoto experimentu bylo vytvořit program, který by předpověděl umístění proteinu v buňce podle aminokyselin v určitém proteinu [4]. Jako trénovací množina byla použita množina 2 427 položek, z nichž každá obsahovala 19 proměnných a 1 výstup. Množina funkcí SMCGP obsahovala všechny sebemodifikující funkce a všechny numerické funkce. Množina funkcí CGP obsahovala pouze matematické operátory. U tohoto typu problému není zřejmé, zda nabízí přístup se sebemodifikací oproti klasickému CGP nějaké výhody. V tabulce 4.3 jsou zobrazeny souhrnné výsledky řešení tohoto problému (založené na 100 bězích každé z reprezentací). CGP i SMCGP dávají podobné výsledky. Zdá se, že přidání sebemodifikace navzdory zvětšení prohledávaného prostoru nijak nebrání vývoji.

4.4 Parita

Cílem tohoto experimentu bylo vygenerovat pomocí SMCGP program, který zjišťuje lichou paritu [6]. Množina funkcí neobsahovala funkci XOR, která obvykle bývá základem obvodů pro výpočet parity. Z logických funkcí byly povoleny pouze AND, OR, NAND, NOR. Hodnota fitness byla určena jako počet správně předpovězených bitů ve všech testovacích případech. Pomocí této funkce se zkoušelo, jestli je program schopen vygenerovat obvody pro paritu různého počtu vstupů.

V první iteraci se zkoušela parita se 2 vstupy, v další se 3 vstupy a tak dále až po maximální počet vstupů. Pokud se kandidátnímu řešení nepodařilo nalézt zcela správné řešení dané velikosti, nebylo již testováno vůči dalším vstupům, což značně šetřilo čas CPU. Fitness funkce byla navržena tak, aby nutila SMCGP k hledání řešení od jednoho testovacího případu k dalšímu. Tento způsob maximalizuje šanci na nalezení obecného řešení.

Tabulka 4.4 obsahuje srovnání výsledků různých metod při řešení tohoto problému. Jsou v ní uvedeny dvě verze SMCGP, přičemž verze označená SMCGP 07 je starší. Z tabulky je patrné, že u menších obvodů je nová verze SMCGP pomalejší, avšak při počtu vstupů větším než 6 je výrazně rychlejší. Důležité je také uvědomit si, že tabulka není úplná,

Tabulka 4.4: Porovnání jednotlivých metod při řešení parity podle [6]

	Průměrně vyhodnocení				**Zrychlení oproti**		
Vstupy	SMCGP	SMCGP 07	CGP	ECGP	SMCGP 07	CGP	ECGP
4	308 643	28 811	81 728	65 296	0,09	0,26	0,21
5	309 990	58 194	293 572	181 920	0,19	0,95	0,59
6	311 022	199 256	972 420	287 764	0,64	3,13	0,93
7	313 489	410 128	3 499 532	311 940	1,31	11,16	1,00
8	313 978	1 080 656	10 949 256	540 224	3,44	34,87	1,72

neboť SMCGP nalezlo řešení až pro 20 vstupů, což by bylo u ostatních metod velmi těžko dosažitelné a není tedy možné v tomto rozsahu metody objektivně porovnat.

4.5 Učení

Cílem tohoto experimentu bylo vytvořit pomocí SMCGP program, který by byl schopný se učit za běhu. Hlavním problémem bylo v tomto případě zajistit, aby učení neprobíhalo během evoluce, ale až při běhu vyvinutého programu. To se podařilo zajistit následujícím způsobem:

Pro učení bylo vybráno všech 16 binárních funkcí se dvěma vstupy. Tyto funkce byly převedeny na pravdivostní tabulky, z nichž 12 jich bylo použito během evoluce a 4 byly ponechány jako testovací. Zde ovšem vyvstává další problém. To, že je někdo schopen se něco naučit (například zpívat) ještě neznamená, že se dokáže naučit i něco zcela jiného (například hrát). Je však velmi pravděpodobné, že schopnost naučit schopnosti k provádění podobných úkonů jsou spojené (například hra na klavír a hra na housle). Autoři práce [5] tedy předpokládali, že učící se systémy by měly být schopny naučit se více úloh (především podobných). Schopnost řešit neznámé problémy se dá tedy za určitých podmínek považovat za učení za běhu.

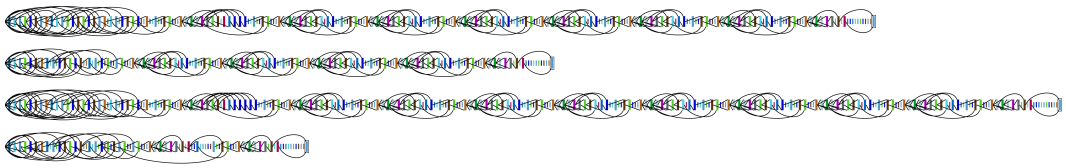
Další možností, jak ověřit schopnost učení, by byla analýza výsledného programu. Vzhledem ke složitosti výsledných programů a jejich špatné čitelnosti pro člověka je však tento způsob poměrně nepoužitelný.

Pravdivostní tabulky pro trénování a testování byly zakódovány pomocí reálných čísel, přičemž binární hodnotě 1 odpovídala hodnota 1.0 a hodnotě 0 odpovídala hodnota -1.0. Po vyhodnocení vzorku se podle znaménka výstupu určí výsledná binární hodnota. Pokud je výsledná hodnota záporná, je výsledkem pravdivostní hodnota 0, jinak 1.

Samotná fitness funkce byla potom určena jako součet odchylek od požadované hodnoty. Jak již bylo řečeno, využívá tato funkce pouze 12 pravdivostních tabulek z celkových 16. Tyto tabulky byly vybrány tak, aby trénovací i testovací množina obsahovaly rovnoměrné rozložení hodnot 0 a 1.

Celkem bylo provedeno 111 experimentů. Z tohoto počtu bylo u 18 experimentů nalezeno řešení, které bylo schopné se naučit 12 trénovacích pravdivostních tabulek. V průměru se řešení naučila správně přibližně 8,6 tabulek. Žádný z vyvinutých programů nebyl schopný se správně naučit všech 16 tabulek. Nejlepší výsledek obsahoval 2 chyby z 16 možných. Nejhorší výsledek obsahoval 9 chyb a v 50% případů byla výstupem pravdivostní hodnota TRUE. To je horší, než náhodný výběr. Řešení, která se naučila všech 12 trénovacích tabulek, obsahovala v testovacích tabulkách v průměru 5,67 chyb.

Na obrázku 4.2 jsou zobrazeny fenotypy pro všechny 4 testovací tabulky. Tyto fenotypy



Obrázek 4.2: Fenotypy pro různé testovací pravdivostní tabulky podle [5]

byly vygenerovány ze stejného genotypu. Jak je z obrázku patrné, jsou délky jednotlivých fenotypů značně odlišné. Odlišná je kromě několika prvních uzlů i jejich struktura. Tyto odlišnosti se generovaly pouze na základě vstupních hodnot programu tak, že v některých případech poskytovaly správné výstupy pro testovací (tedy během evoluce neznámé) tabulky. To lze považovat za učení za běhu.

Ačkoliv výsledky nebyly zcela úspěšné, byla prokázána možnost tvorby učících se programů pomocí SMCGP. Vývoj v této oblasti stále pokračuje a lze tedy v blízké době očekávat další zlepšení.

Kapitola 5

Modifikace algoritmu SMCGP

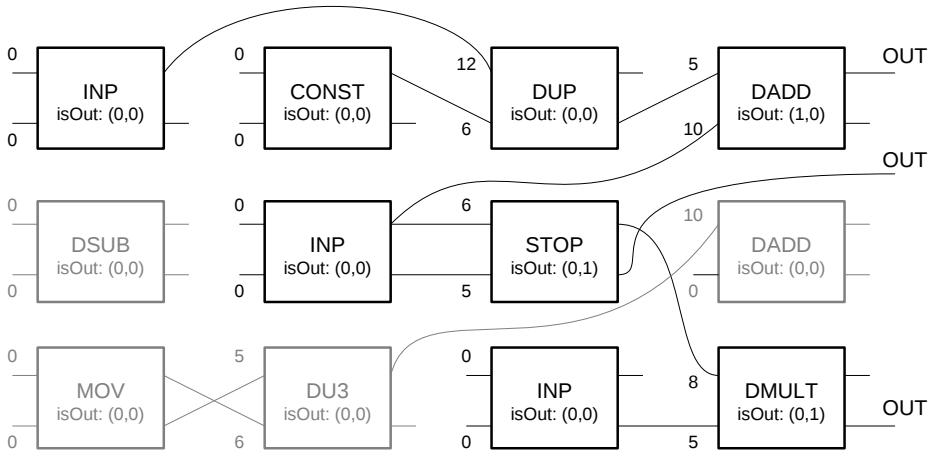
SMCGP je nová metoda, která zatím není příliš prozkoumána. Vzhledem k tomuto faktu tedy existuje mnoho možných vylepšení algoritmu SMCGP, které mohou vést k lepším výsledkům. V této části tedy budou popsány změny provedené oproti původnímu algoritmu SMCGP popsanému v článku [4]. Některé z těchto změn byly zamýšleny od začátku, jiné byly začleněny až v průběhu experimentování s cílem urychlit nebo jiným způsobem vylepšit průběh evoluce.

5.1 Vyhodnocování vstupů

V původní implementaci se dle dostupných informací pravděpodobně vyhodnocují všechny vstupy uzlu. Uvažme, že chromozom může obsahovat i funkce, které využívají méně vstupů, než je počet vstupů uzlu (například funkce identity, negace apod.) Při použití původního přístupu se však zbytečně vyhodnocují také uzly, jejichž výstup se nepoužije. Mohlo by zdát, že se jedná pouze o drobnost. Pokud si však uvědomíme, že k vyhodnocení takového vstupu je třeba vyhodnotit nejen uzel, ke kterému je připojen, ale i všechny uzly, na kterých výstup tohoto uzlu závisí, zjistíme, že počet zbytečně vyhodnocovaných uzlů roste exponenciálně. Nebudou-li se tyto vstupy vyhodnocovat, dojde k poměrně značnému urychlení vyhodnocení chromozomu.

Provedl jsem tedy nepatrnou změnu implementace při vyhodnocování uzlů. Každé funkci lze přiřadit využité vstupy. Toto přiřazení je implementováno dvojrozměrným polem booleovských hodnot, ve kterém každé funkci přísluší jeden řádek a každému vstupu jeden sloupec. Výhodou tohoto přístupu je například fakt, že lze nastavit použití konkrétních vstupů (například prvního, pátého a osmého). V případě uzlů určitých typů, jako jsou například logické funkce, konkrétní výběr vstupů výsledek téměř neovlivní. Nezáleží totiž na tom, zda bude například hradlo NOT negovat první nebo druhý vstup. Připojení vstupní hodnoty na patřičný vstup zajistí evoluce. Pokud však mají být v chromozomu použity funkce uzlů, u kterých na číslech vstupů záleží, je implementace velmi jednoduchá.

I když se neaktivní vstupy nevyhodnocují, při evoluci se nijak neodlišují od ostatních. Během evoluce se neaktivní vstupy mohou přepojovat, což je důležité především z hlediska zachování neutrálních mutací. Důležitost neutrálních mutací je popsána v článku [1].



Obrázek 5.1: Příklad reprezentace chromozomu pomocí modifikované varianty SMCGP

5.2 Uzly s více výstupy

Vzhledem k plánované implementaci evoluce řadicích sítí jsem se rozhodl rozšířit původní model o možnost, aby měl každý uzel grafu více výstupů. Nabízelo se několik různých řešení, například i možnost, aby se počty výstupů jednotlivých uzlů mohly lišit. Tento přístup by sice zvýšil variabilitu systému, ale značně by zpomalil běh programu, protože například relativní indexy uzlů by se v takovém případě musely počítat po jednom. Rozhodl jsem se tedy ponechat konstantní (ale konfigurovatelný) počet výstupů uzlu. U funkcí, které mají výstupů méně, lze nastavit různé chování. Například pro funkce s jedním výstupem lze ostatní výstupy na nulovou hodnotu. Dle mého názoru je však lepším řešením duplikovat výstupní hodnotu na všechny výstupy, protože se tím zvýší konektivita. Příklad chromozomu s takovými uzly je zobrazen v obrázku 5.1. Neaktivní uzly jsou v tomto obrázku zobrazeny šedou barvou.

S touto změnou souvisí i mírně odlišný způsob reprezentace genu, neboť je třeba určovat, který z výstupů uzlu se má použít pro výstup chromozomu. Místo původní booleovské hodnoty určující, zda se jedná o koncový (výstupní) uzel, se ve struktuře genu nachází pole booleovských hodnot o velikosti rovné počtu výstupů.

Používání více výstupů s sebou však nese také nevýhodu ve formě zvětšení prohledávaného prostoru, neboť každý vstup lze v tomto případě připojit na $n$krát více výstupů než u původní varianty, kde n je počet výstupů uzlu. Toto zvětšení stavového prostoru je velmi výrazné a je třeba pečlivě zvážit, zda je v daném případě použití více výstupů opravdu nutné.

Na druhou stranu může v některých případech použití více výstupů urychlit nalezení řešení. Například u řadicích sítí se při klasické reprezentaci s jedním výstupem musí komparátor realizovat dvěma samostatnými uzly MIN a MAX (resp. AND a OR v závislosti na reprezentaci vstupů). Evoluce tedy musí vygenerovat dva uzly, které jsou připojené ke stejným vstupům a potom patřičným způsobem připojit jejich výstupy. Při použití uzlů zastupujících komparátory se dvěma výstupy, kde jeden z výstupů je minimem a druhý maximem, musí evoluce najít pouze jeden uzel s příslušnými vstupy. Další nespornou vý-

hodou tohoto přístupu je snáze pochopitelný výstupní chromozom, ve kterém jsou vidět přímo komparátory.

5.3 Dvojrozměrné pole uzlů

Tradiční CGP využívá běžně dvojrozměrné pole. Oproti tomu původní varianta SMCGP používá pouze jednorozměrné pole. Pokud je chromozom reprezentován jako jednorozměrné pole, může při vhodném nastavení parametru L-back reprezentovat stejný chromozom jako dvojrozměrné pole. Stavový prostor je však větší, protože u dvojrozměrného pole nelze uzly připojovat k uzlům ve stejném sloupci. Toto omezeni u jednorozměrného pole pochopitelně neplatí a počet možných variací propojení uzlů je tedy vyšší. Reprezentace jedince dvojrozměrným polem by tedy mohla urychlit hledání řešení.

Jedním z možných důvodů, proč je SMCGP ve své původní podobě jednorozměrné, jsou právě sebemodifikující funkce. Při jejich použití se totiž může počet genů chromozomu změnit v zásadě o libovolnou hodnotu. Po takové změně však již nemusí být počet genů vhodný pro uspořádání do dvojrozměrného pole požadované velikosti. Tento problém je vyřešen tak, že se při provádění sebemodifikujících funkcí, které mění počet genů, nastavují indexy genů tak, aby se kopírovaly celé sloupce.

Tento přístup zajišťuje, že počet genů zůstane zachovaný tak, aby je bylo možné uspořádat do dvojrozměrného pole o pevně daném počtu řad. Tímto způsobem se mohou kopírovat celé funkční bloky obsažené v rámci jednoho sloupce. To je výhodné v případech, kdy má být výsledná struktura vrstvená tak, že každý sloupec využívá výstupy předchozího sloupce. Na druhou stranu tento přístup zamezuje sebemodifikujícím změnám uvnitř sloupce. Tyto změny tak mohou být realizovány pouze mutacemi během evoluce.

Příklad chromozomu reprezentovaného dvojrozměrným polem uzlů je na obrázku 5.1. Relativní adresa vstupu udává, o kolik výstupů zpět je daný vstup připojen. Do této hodnoty se započítávají i výstupy uzlů v témže sloupci. I když k výstupům uzlů ležícím ve stejném sloupci nelze vstupy uzlu připojit, je tato reprezentace výhodná, protože takový chromozom lze poté snadno převést na jednorozměrnou reprezentaci pouhým zápisem uzlů po jednotlivých sloupcích. Propojení uzlů zůstane v tomto případě zachováno, tudíž bude mít chromozom stále stejnou funkci.

5.4 Nové funkce uzlů

Dalším rozšířením původního modelu je přidání nových funkcí. K sebemodifikujícím funkcím jsem přidal funkci INC. Pokud je uzel s touto funkcí aktivován, zvýší se hodnota jeho parametru P_0 o jedničku. Díky tomu může tento uzel počítat například počet iterací provedených během evoluce. Údaj o pořadí aktuální iterace může být velmi užitečný při evoluci řešení určitých problémů, u kterých je výsledek závislý na pořadí iterace.

Dalšími přidanými funkcemi jsou PRED a SUCC. Tyto funkce nepatří mezi sebemodifikující funkce. Využívají jeden vstup a jejích výstupem je předchůdce, resp. následovník vstupu. Pokud se tato funkce během evoluce duplikuje a zůstane vhodným způsobem propojená s funkcí z předchozí iterace, zvyšuje se její výstup lineárně v závislosti na vzdálenosti od původního uzlu. Toho lze s výhodou využít u experimentů, jejichž výstupy v jednotlivých iteracích jsou závislé na pořadí iterací.

Dalším využitím těchto funkcí je možnost realizovat pomocí nich počítané cykly. To si lze představit například tak, že na počátku může být uzel s konstantou, za ním opakující se

bloky duplikované v každé iteraci. Při vhodném propojení bloků se bude vstupní hodnota duplikačního bloku během iterací snižovat, až se po určitém konečném počtu iterací určeném konstantou na počátku duplikace zastaví.

Poslední novou funkcí je funkce EXP. Jedná se o sebemodifikující funkci, jejíž zkratka pochází ze slova expand. Tato funkce funguje stejně jako sebemodifikující funkce DUP s tím rozdílem, že odstraní uzel v místě vložení duplikovaných uzlů. V zásadě se tedy jedná o nahrazení jednoho uzlu v místě vložení skupinou duplikovaných uzlů. Důvodem přidání této funkce byl předpoklad, že při rozšiřování řadicích sítí o další vstup je vhodné nahradit jeden ze vstupních uzlů původní sítě skupinou nových uzlů. Podrobnosti jsou uvedeny v části věnované experimentu s řadicími sítěmi.

Kapitola 6

Implementace

K implementaci jsem si vybral jazyk C. Důvodem byl především požadavek na co nejvyšší rychlost. V jazyce C++ by byla implementace pochopitelně mnohem přímočařejší a jednodušší, avšak objektová reprezentace značně zpomaluje běh programu. V zájmu co nejvyšší rychlosti je program napsán tak, aby se co nejvíce omezily časově náročné úkony. Kód je tedy minimálně členěn, aby se zamezilo nadbytečnému volání funkcí. Taková struktura ovšem znesnadňuje orientaci v kódu. Tento problém jsem se pokusil částečně řešit používáním maker, která se vyhodnotí již při překladu. Čitelnost programu se tím zvýšila a výkon přitom nebyl ovlivněn. V následující části jsou popsány základní prvky konkrétní implementace.

6.1 Datové typy

Program standardně pracuje se dvěma základními datovými typy. Pro celočíselné hodnoty (například indexy) se používají hodnoty typu unsigned int. Reálná čísla jsou vyjadřována pomocí datového typu double.

Vzhledem k tomu, že pro různé experimenty mohou být nutné různé datové typy proměnných pro vstupy a výstupy, lze požadovaný typ určit pomocí definice `VAR_TYPE`. Je možné zadat libovolný základní datový typ. Pro práci se složitějšími datovými strukturami je možné jako typ použít ukazatel na danou strukturu. V takovém případě je pochopitelně třeba upravit kód pro jednotlivé funkce uzlů, protože jinak by například výsledkem sčítání byl součet ukazatelů, tedy nesprávná hodnota, která by při zpětné interpretaci na strukturu způsobila selhání v důsledku pokusu o přístup k paměti, která není procesu vyhrazena.

Toto omezení by bylo možné eliminovat přepsáním programu do jazyka C++ a použitím přetížených operátorů. Takovou úpravu jsem se však rozhodl neimplementovat z důvodu zachování rychlosti běhu programu.

6.2 Konstanty

V souboru SMCGPdefs.h jsou všechny konstanty určující nejdůležitější parametry reprezentace jedinců a průběhu evoluce. Význam konstant je popsán v následujícím přehledu:

GENE_IN_COUNT	Počet vstupů genu
GENE_OUT_COUNT	Počet výstupů genu
CHROMOSOME_IN_COUNT	Počet vstupů chromozomu
CHROMOSOME_OUT_COUNT	Počet výstupů chromozomu
VAR_TYPE	Typ použitých proměnných
COL_COUNT	Počet sloupců
ROW_COUNT	Počet řádků
L_BACK	Parametr L-back
LAMBDA	Parametr λ pro evoluční strategii
MUTATE_MAX	Maximální počet mutací v rámci chromozomu
NON_SELFMOD_PERC	Procentní podíl jiných než sebemodifikujících funkcí
MUTATE_FUNC_PERC	Pravděpodobnost (v procentech), že se při mutaci změní funkce uzlu
MUTATE_INPUT_PERC	Pravděpodobnost (v procentech), že se při mutaci změni připojení vstupu uzlu
MUTATE_PARAM_PERC	Pravděpodobnost (v procentech), že se při mutaci změní jeden z reálných parametrů genu
GENERATION_COUNT	Maximální počet generací, po které se má hledat řešení

Ve výpisu jsou uvedené konstanty udávající pravděpodobnost různých změn během mutace. Při mutaci se může kromě těchto změn vyskytnout ještě změna nastavení aktivity výstupů. Tato pravděpodobnost není v přehledu uvedena, protože se dopočítává jako 100% - součet ostatních pravděpodobností pro mutaci.

Konstanta NON_SELFMOD_PERC udává pravděpodobnost v procentech, že se při generování funkce uzlu vygeneruje funkce, která není sebemodifikující. To může být vhodné v případech, kdy je sebemodifikujících funkcí výrazná převaha. Úpravou tohoto parametru lze zajistit, že se v nově generovaných chromozomech budou sebemodifikující a ostatní funkce vyskytovat v požadovaném poměru.

6.3 Reprezentace chromozomu

Chromozom je reprezentován strukturou:

```
typedef struct {
        SMCGP_GENE *firstGene;
        UINT geneCount;
        double fitness;
        BOOL wasParent;
    } SMCGP_CHROMOSOME;
```

Význam jednotlivých prvků struktury je následující:

`firstGene`	Tento prvek obsahuje ukazatel na místo v paměti, kde jsou uloženy geny chromozomu.
`geneCount`	Obsahuje údaj o počtu genů v chromozomu. Tato hodnota je důležitá především pro určení rozsahu dat genů v paměti. Není tedy vhodné ji měnit, aniž by se patřičným způsobem změnila data genů.
`fitness`	Tato položka slouží k uložení vypočtené hodnoty fitness, aby s ní bylo možné pracovat v libovolné funkci, do které je chromozom předán. Údaje o hodnotách fitness by samozřejmě bylo možné uchovávat mimo chromozom, ale tento způsob se mi zdál nejvhodnější z hlediska snadného použití.
`wasParent`	Tato proměnná udává, jestli byl chromozom v předchozím kole vybrán jako rodič. Díky tomuto údaji lze zajistit neutrální prohledávání.

6.4 Reprezentace genu

Gen je reprezentován touto strukturou:

```
typedef struct {
    UINT inputs[GENE_IN_COUNT];
    FUN_TYPE function;
    BOOL isOutput[GENE_OUT_COUNT];
    float P[3];
} SMCGP_GENE;
```

Položky této struktury mají následující význam:

`inputs`	Relativní adresy uzlů připojených k jednotlivým vstupům. Relativní adresa udává, o kolik výstupů zpět je tento vstup připojen.
`function`	Obsahuje údaj o funkci genu.
`isOutput`	Udává, zda jsou jednotlivé výstupy tohoto genu výstupy chromozomu. Pokud je některá z hodnot pole nastavena na hodnotu true, neznamená to ještě, že se tento výstup opravdu použije jako výstup chromozomu.
`P`	Parametry genu. Tyto hodnoty se využívají při provádění sebemodifikujících funkcí.

6.5 Funkce pro manipulaci s chromozomem

V této části jsou popsány funkce, které lze použít pro manipulaci s chromozomem. Tyto funkce jsou implementovány tak, aby do nich nebylo při návrhu nového experimentu nutné zasahovat. Jedinou výjimkou jsou případy, kdy je nutné v rámci experimentu přidat další sebemodifikující funkce.

```
SMCGP_CHROMOSOME * GenerateRandomChromosome();
```

Tato funkce generuje nový chromozom. Není nutné jí předávat žádné parametry, protože všechny hodnoty potřebné k vytvoření nového chromozomu jsou definovány výše popsanými konstantami. Návratovou hodnotou je ukazatel na vygenerovaný chromozom, případně NULL, pokud se nepodaří alokovat paměť.

Samotné generování probíhá tak, že se nejprve podle definovaných konstant vytvoří chromozom. Následně se vytvoří pole obsahující geny chromozomu. Generování genů probíhá náhodně, avšak je možné jej ovlivnit různými parametry. Tyto parametry většinou není nutné měnit, i když v jistých případech může jejich nastavení přinést zlepšení výsledků evoluce.

```
SMCGP_CHROMOSOME * CopyChromosome(SMCGP_CHROMOSOME * origChrom);
```

Pomocí této funkce lze vytvořit kopii chromozomu. Při vytváření kopie se vytvoří i kopie genů, takže lze na kopii provádět libovolné úpravy, aniž by jimi byl originál jakkoliv ovlivněn. Vstupním parametrem je ukazatel na kopírovaný chromozom. Výstupem je ukazatel na kopii chromozomu, případně NULL, pokud se vyskytne chyba.

```
void DestroyChromosome(SMCGP_CHROMOSOME * chrom);
```

Tato funkce slouží k bezpečnému odstranění chromozomu. Během odstranění chromozomu se patřičným způsobem uvolní i paměť alokovaná pro geny tohoto chromozomu.

```
void MutateChromosome(SMCGP_CHROMOSOME * chromosome);
```

Na chromozomu předaném do této funkce se provede mutace v závislosti na patřičných konstantách a pravděpodobnostech.

```
void EvaluateChromosome(SMCGP_CHROMOSOME * chromosome,
VAR_TYPE * inputs,
VAR_TYPE *outputs,
UINT* inputOrder,
UINT* outputOrder,
BOOL* usedBlocks,
UINT *toDoList);
```

Tato funkce provádí vyhodnocení výstupů chromozomu v závislosti na daných vstupech. Poměrně vysoký počet parametrů je způsoben tím, že tato funkce má za úkol poskytnout o vyhodnocení chromozomu co nejvíce informací. Přehled významu parametrů se nachází v následujícím seznamu.

`chromosome`	Ukazatel na vyhodnocovaný chromozom
`inputs`	Ukazatel na pole vstupů
`outputs`	Ukazatel na pole výstupů
`inputOrder`	Ukazatel na pole pro uložení informace o pořadí vyhodnocení uzlů s funkcí INP
`outputOrder`	Ukazatel na pole, do kterého se uloží informace o tom, které uzly byly použity jako výstupní a v jakém pořadí byly vyhodnocovány.
`usedBlocks`	Ukazatel na pole pro uložení informace o využitých uzlech
`toDoList`	Ukazatel na seznam sebemodifikujících operací, které byly při vyhodnocování chromozomu aktivovány

Pole pro uložení informací o pořadí vstupů a výstupů se používají především ve funkcích pro vizualizaci chromozomu. Lze je však použít například také k analýze atd. Informace o využitých uzlech může být součástí vyhodnocení fitness funkce, aby byla zvýhodněna řešení obsahující co nejmenší počet aktivních uzlů.

Do pole toDoList se uloží indexy uzlů sebemodifikujících funkcí aktivovaných při vyhodnocování chromozomu. Aby byl uzel přidán do tohoto seznamu, je nutné, aby byl při vyhodnocování aktivní a navíc, aby jeho druhý vstup byl větší než první vstup. Jinak se uzel do seznamu nepřidá. Přidání uzlu do seznamu neznamená jeho provedení, protože chromozom můžeme chtít vyhodnotit vícekrát, aniž by se měnil. K provedení změn slouží následující funkce.

```
SMCGP_CHROMOSOME* SelfModification(SMCGP_CHROMOSOME * source, UINT * toDo);
```

Tato funkce vykonává sebemodifikující funkce uzlů obsažených v seznamu toDo. Vstupními parametry jsou zdrojový chromozom a seznam uzlů, jejichž sebemodifikační funkce se má provést. Výstupní hodnotou je ukazatel na nový chromozom vzniklý provedením sebemodifikujících funkcí.

6.6 Funkce pro vizualizaci chromozomu

V této části jsou uvedeny funkce, které lze použít k vizualizaci chromozomu. Tyto funkce slouží ke zobrazení chromozomu v podobě vhodné pro prezentaci výsledků nebo jejich další použití. Všechny zde uvedené funkce existují ve dvou provedeních, s prefixem Print a prefixem Log. Funkce s prefixem Print vypisují výsledek na obrazovku, s prefixem Log do souboru. Funkce s prefixem Print neobsahují jako vstupní parametr soubor. Jinak je jejich použití stejné a nebudou tedy popisovány samostatně.

```
void LogCode(FILE * logFile, SMCGP_CHROMOSOME * chromosome);
```

Pomocí této funkce lze vypsat chromozom ve tvaru znovupoužitelného úseku kódu v jazyce C. Takový výstup lze následně začlenit do programu a pracovat s ním. Tento přístup je vhodný v případech, kdy chcete nalezený výsledek evoluce krokovat, abyste pochopili, případně ověřili jeho činnost.

Vstupem funkce je ukazatel na výstupní soubor a ukazatel na chromozom, který se má vypsat. Výstup se vypíše v následujícím tvaru:

```
SMCGP_CHROMOSOME test;
SMCGP_GENE genes[20];

test.geneCount = 20;
test.firstGene = &genes[0];

genes[0].inputs[0] = 0;
genes[0].inputs[1] = 0;
genes[0].isOutput[0] = true;
genes[0].isOutput[1] = true;
genes[0].function = 2;
genes[0].isEvaluated = false;
genes[0].P[0] = 1.118770
genes[0].P[1] = 1.635329
genes[0].P[2] = -0.185113
genes[1].inputs[0] = 0;
genes[1].inputs[1] = 0;
```

```
genes[1].isOutput[0] = true;
genes[1].isOutput[1] = true;
genes[1].function = 6;
genes[1].isEvaluated = false;
genes[1].P[0] = 0.444992
genes[1].P[1] = 2.345152
genes[1].P[2] = -0.689771
genes[2].inputs[0] = 0;
genes[2].inputs[1] = 0;
genes[2].isOutput[0] = true;
genes[2].isOutput[1] = true;
genes[2].function = 7;
genes[2].isEvaluated = false;
genes[2].P[0] = 0.300082
genes[2].P[1] = -1.800263
genes[2].P[2] = 1.096927
...
```

Tento výstup pochopitelně pokračuje dále. Zde je však uveden pouze pro ilustraci a je tedy vypsáno jen několik prvních genů.

```
void LogCGPView(FILE * logFile, SMCGP_CHROMOSOME * chromosome);
```

Tato funkce vypíše chromozom ve tvaru vhodném pro zobrazení programem CGPView. Tento program byl vytvořen na Fakultě informačních technologií VUT v Brně a je navržen pro práci s výsledky tradičního CGP. Nepokrývá tedy všechny možnosti SMCGP. Díky jeho značné univerzalitě jej však lze po úpravách použít i pro SMCGP. Hlavní výhodou tohoto programu jsou funkce pro detekci a eliminaci zbytečných hradel.

Pomocí této funkce je možné vypisovat i chromozomy obsahující geny s více výstupy. S takovými geny však CGPView neumí pracovat. Aby takový chromozom bylo možné v CGPView zobrazit, vypisují se uzly s více výstupy jako několik uzlů umístěných pod sebou, každý s jedním výstupem.

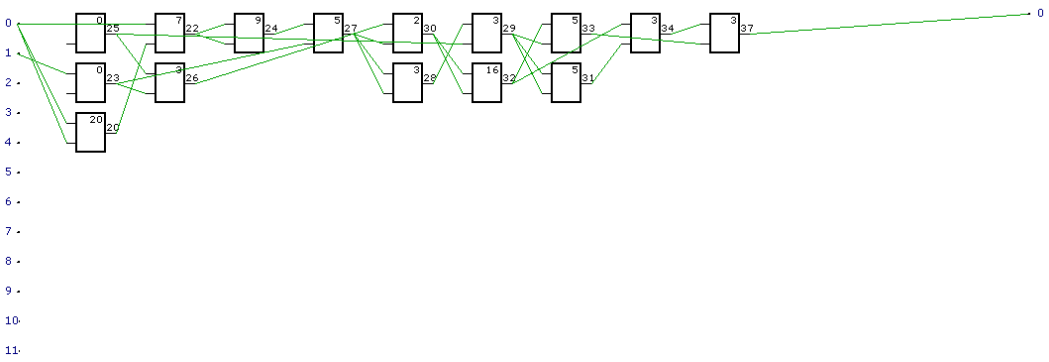
Vstupem této funkce je ukazatel na soubor, do kterého se má výstup vypsat a ukazatel na chromozom, který se má vypsat. Výstup této funkce má následující tvar:

```
{20,1, 20,1, 2,4,0}([20]0,0,20)([21]20,20,13)([22]0,20,7)([23]1,0,0)
([24]22,22,9)([25]0,0,0)([26]25,23,3)([27]24,23,5)([28]27,27,3)([29]28,25,3)
([30]26,27,2)([31]29,29,5)([32]30,30,16)([33]32,29,5)([34]32,31,3)
([35]31,34,3)([36]35,34,4)([37]34,33,3)([38]36,35,2)([39]38,38,18)(37)
```

Pokud takový výstup zadáme do CGPView a provedeme eliminaci zbytečných a automatické rozmístění zbývajících hradel, dostaneme výstup podobný jako na obrázku 6.1.

```
void LogDot(FILE * logFile, SMCGP_CHROMOSOME * chromosome);
```

Při implementaci rozšíření umožňujícího, aby uzly obsahovaly více výstupů, jsem musel čelit problému, jak takový chromozom vypsat. Nejprve jsem používal nástroj CGPView a uzly rozdělené na více uzlů podle počtu vstupů. Časem se však ukázalo, že tento přístup není příliš pohodlný, protože pro každou sadu funkcí genů je nutné použít v programu



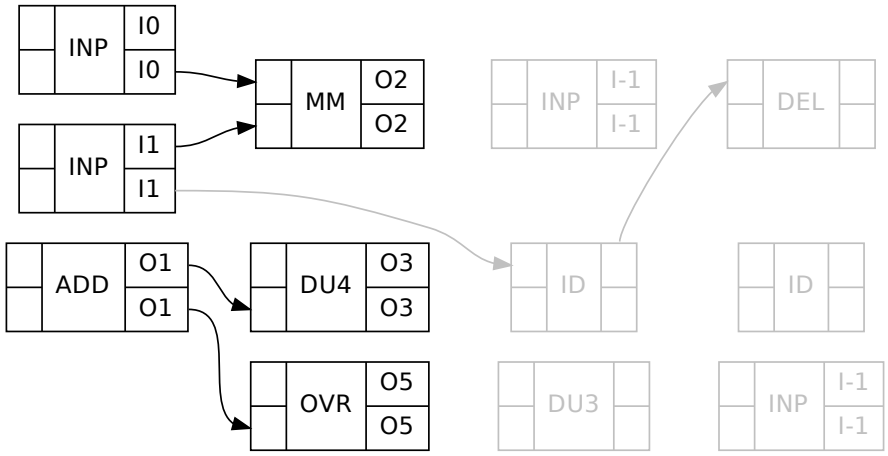
Obrázek 6.1: Znázornění chromozomu pomocí programu CGPView

CGPView jinou konfiguraci. Vzhledem k tomu, že sada funkcí genů se může poměrně často měnit, rozhodl jsem se implementovat vlastní vizualizaci.

Nejprve jsem uvažoval o tom, že implementují vlastní jednoduché zobrazovaní s výstupem do některého z vektorových formátů. Zajištění správného a dobře čitelného vykreslení grafu chromozomu je však dost složitý úkol, tak jsem se nakonec rozhodl využít program graphViz, který je k vykreslování grafů přímo určen.

Tato funkce tedy vypíše chromozom zapsaný v jazyce DOT. Tento výstup lze potom zpracovat pomocí součástí programu graphViz a exportovat výsledek do široké škály různých formátu, ať již vektorových nebo rastrových. Výstup této funkce může vypadat například takto:

```
digraph smcgp {
        rankdir=LR
        {rank = same;
        node0 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|INP|{<out0>I1|<out1>I1}}"]
        node1 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|ADD|{<out0>O1|<out1>O1}}"]
        node2 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|INP|{<out0>I0|<out1>I0}}"]
        }
        {rank = same;
        node3 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|MM|{<out0>O2|<out1>O2}}"]
        node4 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|DU4|{<out0>O3|<out1>O3}}"]
        node5 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|OVR|{<out0>O5|<out1>O5}}"]
        }
        {rank = same;
        node6 [color=gray,fontcolor=gray,shape=record,
                label="{{<in0>|<in1>}|INP|{<out0>I-1|<out1>I-1}}"]
        node7 [color=gray,fontcolor=gray,shape=record,
                label="{{<in0>|<in1>}|ID|{<out0>|<out1>}}"]
        node8 [color=gray,fontcolor=gray,shape=record,
```



Obrázek 6.2: Znázornění chromozomu pomocí programu graphViz

```
        label="{{<in0>|<in1>}|DU3|{<out0>|<out1>}}"]
    }
    {rank = same;
    node9 [color=gray,fontcolor=gray,shape=record,
        label="{{<in0>|<in1>}|DEL|{<out0>|<out1>}}"]
    node10 [color=gray,fontcolor=gray,shape=record,
        label="{{<in0>|<in1>}|ID|{<out0>|<out1>}}"]
    node11 [color=gray,fontcolor=gray,shape=record,
        label="{{<in0>|<in1>}|INP|{<out0>I-1|<out1>I-1}}"]
    }
    edge [style=invis]
    node0->node3->node6->node9
    node1->node4->node7->node10
    node2->node5->node8->node11
    node0->node1->node2
    edge [style=solid]
    node2:out1->node3:in0 [color=black, weight=0]
    node0:out0->node3:in1 [color=black, weight=0]
    node1:out0->node4:in1 [color=black, weight=0]
    node1:out1->node5:in1 [color=black, weight=0]
    node0:out1->node7:in0 [color=gray, weight=0]
    node7:out0->node9:in0 [color=gray, weight=0]
}
```

Výsledkem zpracování uvedeného kódu pomocí programu graphViz je obrázek 6.2.

Menším problémem při tomto druhu zobrazování je skutečnost, že graphViz umožňuje zarovnávat uzly pouze v jednom směru. Ve druhém směru lze zarovnání do určité míry vynutit použitím neviditelných hran s vyšší vahou, avšak ne vždy je zajištěn zcela správný výsledek a uzly tak mohou být vůči očekávané poloze posunuty.

6.7 Problémy při implementaci

Při implementaci modelu SMCGP jsem narazil na několik nejasností. Asi nejpodstatnější z nich byla absence jakékoliv zmínky o generování hodnot parametrů genů. Vzhledem k tomu, že jsou tyto parametry využívány převážně jako indexy ohraničující oblast měněných genů při sebemodifikujících funkcích, rozhodl jsem se je implementovat následujícím způsobem. Při generování chromozomu se hodnoty parametrů nastaví na náhodnou hodnotu s normálním rozložením, přičemž $\sigma =$ `COL_COUNT`. Díky tomu by měly být parametry genů dostatečně vysoké, aby mohla sebemodifikace působit i na větší vzdálenosti.

Kapitola 7

Experimentální ověření modifikované metody SMCGP

V této kapitole budou popsány experimenty provedené pomocí vytvořené implementace SMCGP. Experimenty jsem se snažil volit tak, aby bylo jasně patrné, jaké výhody nabízí SMCGP oproti CGP, případně i jiným evolučním technikám. U každého experimentu je uvedeno nastavení, se kterým byl experiment prováděn. Díky tomu je možné experimenty reprodukovat. Vedlo mě k tomu především to, že v literatuře, ze které jsem čerpal, je většinou uveden pouze výčet použitých funkcí a žádné další parametry, takže není možné přímé srovnání, neboť i malé změny některých parametrů zcela mění průběh evoluce. Parametry evoluce (velikost populace, četnost mutací atd.) byly voleny tak, aby bylo možné srovnat výsledky s existujícími experimenty. Pokud neexistoval žádný experiment, se kterým by bylo možné výsledky srovnat, byly parametry nastaveny na hodnoty poskytující nejlepší výsledky. Jejich hodnoty byly v takovém případě určeny experimentálně.

7.1 Druhé mocniny

Jako první experiment jsem si zvolil výpis druhých mocnin. Cílem tohoto experimentu je vyvinout pomocí evoluce chromozom, který bude dávat na výstupu druhé mocniny přirozených čísel. Při každé iteraci má být na výstupu jedna hodnota, tedy při první iteraci hodnota 1, při druhé 4, při třetí 9 atd.

7.1.1 S využitím násobení

Nejprve jsem se pokusil evolvovat chromozom, který využívá všechny sebemodifikující funkce a z běžných funkcí DADD, DMULT, INP a NOP. Pro běh programu jsem zvolil následující hodnoty:

Tabulka 7.1: Shrnutí výsledků hledání druhých mocnin při použití funkce DMULT

Prům. poč. vyh.	Min. poč. vyh.	Max. poč. vyh	Obecná řešení
355 825	50	2 000 000	86%

GENE_IN_COUNT	2
GENE_OUT_COUNT	1
CHROMOSOME_IN_COUNT	1
CHROMOSOME_OUT_COUNT	1
VAR_TYPE	UINT
COL_COUNT	20
ROW_COUNT	1
L_BACK	8
LAMBDA	4
MUTATE_MAX	2
MAX_SELFMOD	2

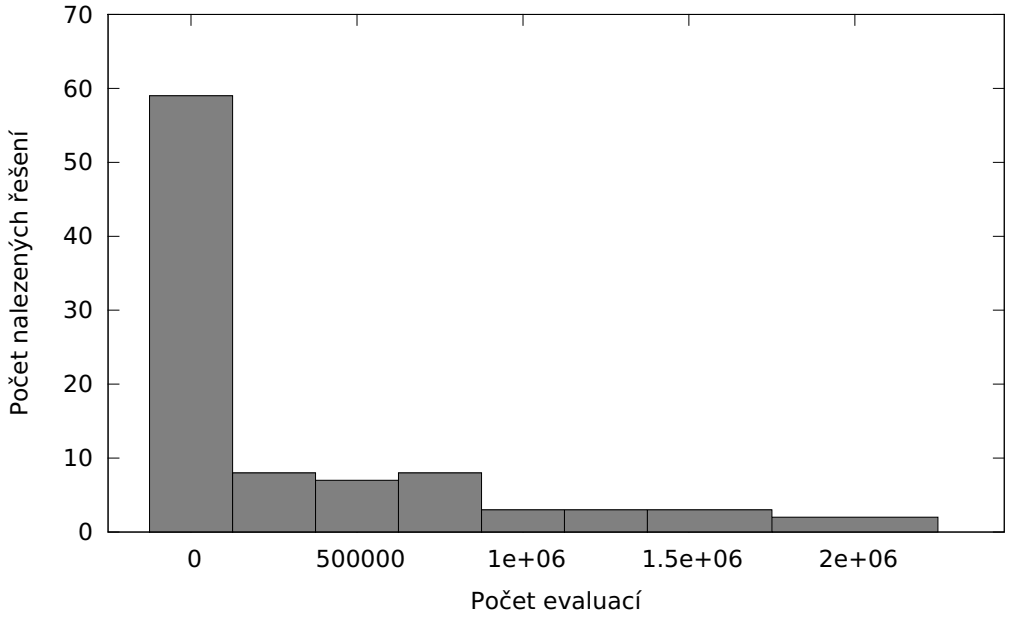
Vzhledem k tomu, že při využití funkce DMULT lze očekávat velmi krátké řešení, zvolil jsem počet sloupců 20. Maximální počet mutací je podle [4] vhodné volit jako 10% počtu genů. Při experimentování s různým nastavením této hodnoty jsem došel k přibližně shodnému závěru. Při použití vyšší hodnoty dochází k velkým změnám chromozomů, což má většinou za následek rozbití již vyevolvovaných stavebních bloků. Pokud je použita nižší hodnota, evoluce konverguje velmi pomalu a velmi často uvázne v lokálním optimu.

Experiment byl spuštěn stokrát. Evoluce probíhala vždy do nalezení řešení nebo dosažení hranice 2,5 milionů evaluací. Při hledání řešení se testovalo, zda chromozom generuje prvních 10 druhých mocnin. Pokud ano, byl tento chromozom přijat jako řešení a následně se testovalo až po stý prvek, zda je řešení obecné. Výsledky této části experimentu dopadly dle očekávání. V naprosté většině případů bylo nalezeno řešení, i když se vyskytly i případy, kdy zůstala evoluce v lokálním optimu. Shrnutí výsledků je v tabulce 7.1.

Jak je z tabulky patrné, bylo obecné řešení nalezeno v 86% případů. To není příliš překvapivý údaj. Zajímavější je rozložení počtu evaluací nutných k nalezení řešení. Toto rozložení je znázorněno v histogramu 7.1.

Z histogramu je zřejmé, že většina řešení je nalezena již během prvních 500 000 evaluací. Podrobným zkoumáním výsledků jsem rovněž zjistil, že téměř všechna obecná řešení se nachází také v tomto rozsahu, zatímco řešení, která nejsou obecná, bývají nalezena v pozdějších evaluacích.

Dalším zajímavým aspektem, na který jsem při řešení tohoto problému narazil, je vliv fitness funkce. Ve výše uvedeném případě vyjadřovala fitness funkce počet správně nalezených členů. Následně jsem upravil fitness funkci tak, aby vyjadřovala vzdálenost mezi skutečným a požadovaným výstupem. Tato funkce se může zdát jako vhodnější, protože má pozvolný průběh na rozdíl od původní funkce, jejíž hodnota se měnila skokem. Ve skutečnosti jsou však výsledky při použití této funkce horší. Řešení bývá nalezeno častěji, ale často není obecné. To si vysvětluji tím, že funkce nutí chromozom k tomu, aby generoval hodnoty co nejbližší hodnotám požadovaným. Díky tomu bývá často nalezeno řešení, které dobře aproximuje prvních deset členů, ale dále pokračuje k jiným hodnotám.



Obrázek 7.1: Histogram počtu nalezených řešení v závislosti na počtu evaluací. Většina řešení je nalezena během prvních 500 000 evaluací.

7.1.2 Bez využití násobení

Předchozí experiment byl důkazem, že SMCGP dokáže generovat obecná řešení. Cílem tohoto experimentu je dokázat, že pomocí SMCGP lze generovat posloupnost druhých mocnin přirozených čísel i bez použití násobení. Jak již bylo uvedeno v kapitole věnované příkladům použití SMCGP, ke generování této řady bez použití násobení je nutné, aby program mohl sám sebe modifikovat. SMCGP takové modifikace umožňuje a mělo by tedy být možné pomocí něj nalézt obecné řešení.

Z použitých funkcí tedy odstraníme násobení a celý experiment zopakujeme. Parametry zůstávají stejné jako v předchozím případě, mění se pouze následující

`COL_COUNT`	`20/40/60/80/100`
`MUTATE_MAX`	`2/4/6/8/10`

Tento zápis znamená, že bylo provedeno více experimentů s různými hodnotami počátečního počtu sloupců a maximálního počtu mutací. Odpovídající dvojice počtu sloupců a mutací jsou v zápisu na stejných pozicích. V předchozím experimentu bylo zjištěno, že většina řešení byla nalezena pod 2 500 000 evaluací. Ověřil jsem, že totéž platí i u tohoto experimentu. Maximální počet evaluací tedy můžeme omezit na 2 500 000, protože to výsledek nijak neovlivní.

Pro každou z pěti možných hodnot počtu sloupců bylo provedeno 100 běhů. Po zkušenostech z předchozího experimentu jsem použil fitness funkci vyjadřující počet správně vygenerovaých prvků. Vyhodnocení a ověření obecnosti probíhá stejným způsobem jako v předchozím případě. Výsledky experimentu jsou vypsány v tabulce 7.2.

Podle výsledků lze usuzovat, že počet sloupců má na průběh evoluce poměrně malý vliv. Výjimkou je počet sloupců 20, který se od průměru výrazně liší. To lze vysvětlit nejmenším prohledávaným prostorem, který umožňuje nalézt řešení nejrychleji. Dalším důvodem

Tabulka 7.2: Shrnutí výsledků hledání druhých mocnin bez použití funkce DMULT

Počet sloupců	Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
20	74 548	380	14	100%
40	145 351	50	17	76,5%
60	122 071	74	18	77,9%
80	114 284	103	19	73,7%
100	103 397	95	18	77,9%

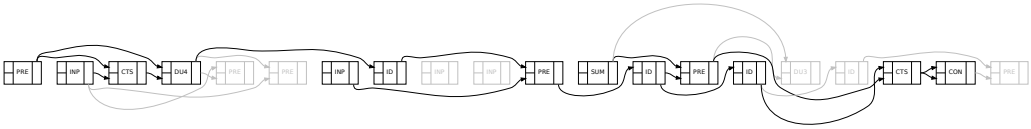
Tabulka 7.3: Porovnání výsledků s výsledky existujícího experimentu

	Prům. poč. vyhodnocení	Min. poč. vyh.	% obecných
Výsledky podle [4]	141 846	392	84,3
Navržená metoda	111 930	140	81,2

může být nízký počet nalezených řešení, protože v tak malém vzorku může průměrnou hodnotu do značné míry ovlivnit i jedna odlehlá hodnota. Po prozkoumání konkrétních výsledků jsem však zjistil, že hodnoty jsou rozptýleny poměrně rovnoměrně, tudíž je výsledek pravděpodobně zapříčiněn nejmenším prohledávaným prostorem.

V tabulce 7.3 je porovnání průměrných hodnot výše uvedených výsledků s výsledky stejného experimentu publikovanými v článku [4].

Z tabulky je zřejmé, že hodnoty přibližně souhlasí. Mírné rozdíly mohou být způsobeny například vyšším počtem maximálních povolených evaluací v článku [4] nebo rozdílnou implementací a nastavením parametrů. Na obrázku 7.2 je zobrazeno obecné řešení nalezené při nastavení počtu sloupců genotypu na hodnotu 20.



Obrázek 7.2: Obecné řešení problému druhých mocnin.

Tabulka 7.4: Výsledky generování Fibonacciho posloupnosti

Vstupy	Počet iterací	Prům. poč. vyhodnocení	% nalezených	% obecných
0 1	10	756 875	82	58,5
0 1	50	752 015	30	46,7
1 2	10	682 360	84	60,7
1 2	50	848 690	32	31,2

7.2 Fibonacciho posloupnost

Dalším typickým příkladem využití SMCGP je generování Fibonacciho posloupnosti. Vzhledem k tomu, že se jedná o iterační posloupnost, měla by být velmi vhodná pro implementaci pomocí SMCGP. Vyzkoušel jsem čtyři různá nastavení podmínek evoluce. Jedním parametrem je počet iterací, které se testují, aby byl chromozom přijat jako řešení (ne nutně obecné). Další testovanou možností je změna vstupů chromozomu. Počet iterací jsem zvolil v jednom případě 10, ve druhém případě 50. Jako vstupy chromozomu jsou použity dvojice (0, 1) a (1, 2).

Ostatní parametry experimentu jsou následující:

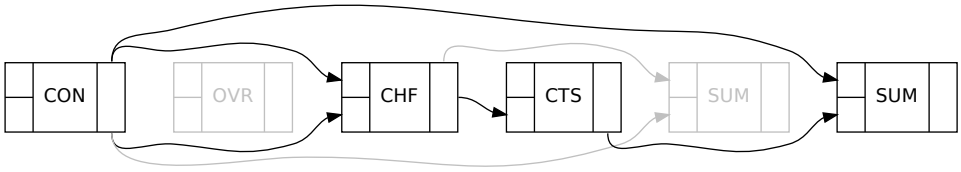
```
GENE_IN_COUNT           2
GENE_OUT_COUNT          1
CHROMOSOME_IN_COUNT     2
CHROMOSOME_OUT_COUNT    1
VAR_TYPE                unsigned long long
COL_COUNT               6
ROW_COUNT               1
L_BACK                  6
LAMBDA                  4
MUTATE_MAX              2
MAX_SELFMOD             2
```

Všimněme si, že počet sloupců je nastaven na pouhých 6. Je tomu tak proto, aby se omezila velikost prohledávaného prostoru. Tuto hodnotu jsem určil sérií pokusů jako nejmenší možný počet sloupců pro uspokojivé generování Fibonacciho posloupnosti v SMCGP.

Byly prověřeny všechny čtyři kombinace vstupních parametrů. Fitness hodnota se počítala jako počet správně určených prvků posloupnosti. Po nalezení řešení byla u každé z kombinací ověřena jeho obecnost až po 93. člen. Počet evaluací byl opět shora omezen, konkrétně hodnotou 5 000 000 evaluací. V tabulce 7.4 jsou vypsány výsledky pro všechny kombinace.

Když výsledné hodnoty porovnáme s hodnotami podobného experimentu publikovanými v [4], zjistíme, že počet evaluací se pro jednotlivé kombinace příliš neliší. Na použitých vstupech evidentně také příliš nezáleží. Rozdílem však je, že při ověřování 50 iterací bylo nalezeno poměrně málo řešení. To přičítám skutečnosti, že u některých běhů byla pravděpodobně dosažena horní hranice evaluací a považovaly se tedy za neúspěšné.

Zajímavějším údajem je poměr obecných řešení, který je také nízký. Po prohlédnutí protokolů běhů se však ukázalo, že v mnoha případech bylo řešením generováno 91 nebo 92 správných hodnot posloupnosti a pouze 1 nebo 2 špatné. Příčinou tohoto jevu by dle mého názoru mohla být aktivace nějakého sebemodifikujícího genu na základě hodnoty jednoho z prvků. Taková aktivace se může bez problémů provést třeba jen v jedné iteraci. V dalších



Obrázek 7.3: První iterace obecného řešení Fibonacciho posloupnosti

iteracích už se tento gen aktivovat nemusí. Vyšší podíl obecných řešení ve srovnávaných hodnotách může být zapřičiněn právě tím, že při kontrole obecnosti se uvažuje pouze prvních 74 prvků.

V [4] jsou popsány i výsledky pokusů o generování Fibonacciho posloupnosti pomocí různých dalších metod, například lineárního GP, objektově orientovaného GP, rekurzivních stromových struktur a dalších. V porovnání s těmito metodami si SMCGP vede velmi dobře. Hlavní výhodou je dle mého názoru, že se řešení může vyvíjet z velmi malé počáteční struktury. Například chromozom jednoho z obecných řešení Fibonacciho posloupnosti získaný navrženou metodou je na obrázku 7.3.

Tabulka 7.5: Shrnutí výsledků hledání funkce faktoriál s využitím funkce DMULT

Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
262 895	1 791	11	100%

7.3 Faktoriál

Cílem tohoto experimentu je implementovat pomocí SMCGP funkci faktoriál. Tuto funkci by vzhledem k tomu, že je možné zapsat ji rekurzivně, mohlo být možné implementovat. Experiment je rozdělen na dvě části. V první části se pokusím o realizaci funkce faktoriál s využitím násobení. Ve druhé části bude násobení zakázáno.

7.3.1 S využitím násobení

V této části experimentu jsou při evoluci ze základních funkcí použity pouze funkce NOP, INP, DADD, DSUB, DMULT, DDIV a CONST. Sebemodifikující funkce jsou použity všechny. Fitness funkce udává počet správných prvků. Vyhodnocuje se prvních 10 iterací. Pokud je nalezeno řešení, kontroluje se obecnost až po faktoriál čísla 20, což je nejvyšší hodnota, kterou je možné uložit do 64bitové proměnné. Nastavení parametrů evoluce je následující:

```
GENE_IN_COUNT           2
GENE_OUT_COUNT          1
CHROMOSOME_IN_COUNT     1
CHROMOSOME_OUT_COUNT    1
VAR_TYPE                unsigned long long
COL_COUNT               15
ROW_COUNT               1
L_BACK                  6
LAMBDA                  4
MUTATE_MAX              2
MAX_SELFMOD             2
```

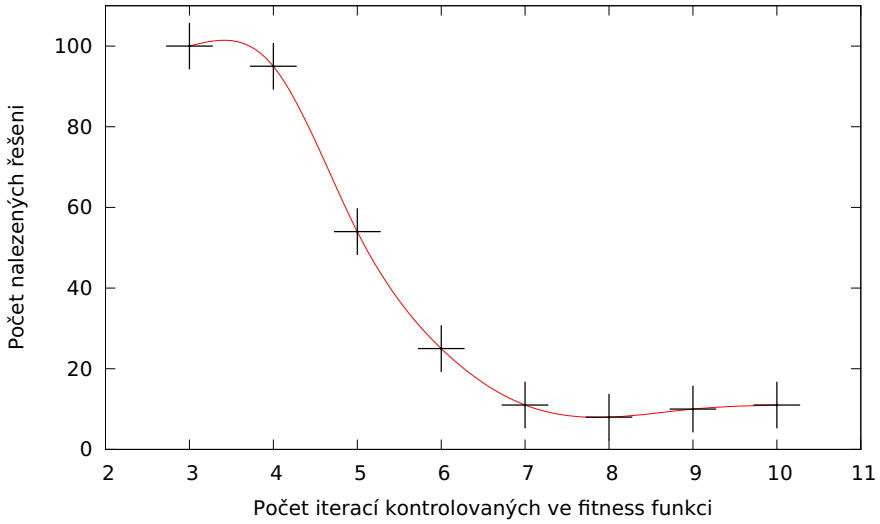
Bylo provedeno 100 běhů, přičemž maximální počet evaluací byl nastaven na hodnotu 1 000 000. Výsledky jsou shrnuty v tabulce 7.5.

Jak je z tabulky jasně patrné, není počet nalezených výsledků příliš vysoký. Je to pravděpodobně způsobeno vyšší složitostí této funkce a nastavením počtu maximálních evaluací. Pokud tuto hranici porovnáme s počty evaluací nutnými k nalezení Fibonacciho řady, nejsou dosažené výsledky příliš překvapivé. Vyšší počet evaluací jsem však nenastavoval, protože potom by evoluce trvala příliš dlouho a výsledky by pravděpodobně nebyly o mnoho lepší. Kromě toho se evoluce v důsledku konvergence často zastaví v nějakém lokálním optimu. Podíl obecných řešení je naproti tomu velmi příznivý.

Řešení tohoto příkladu mě však přivedlo na myšlenku porovnat, jak na sobě závisí počet kontrolovaných iterací a podíl obecných řešení. Provedl jsem tedy několik dalších experimentů s počty kontrolovaných iterací od 3 do 7. Parametry, fitness funkce i maximální počet evaluací byly u těchto běhů stejné jako v předchozím případě. Pro každou variantu bylo provedeno 100 běhů. Výsledky jsou vypsány v tabulce 7.6.

Tabulka 7.6: Shrnutí výsledků hledání funkce faktoriál s využitím funkce DMULT

Počet iterací	Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
3	93 850	157	100	0%
4	230 999	59	95	2,1%
5	405 260	68	54	13%
6	359 961	1 570	25	44%
7	191 600	863	11	63,6%
8	308 321	5 663	8	87,5%
9	309 518	2 106	10	90%
10	262 895	1 791	11	100%

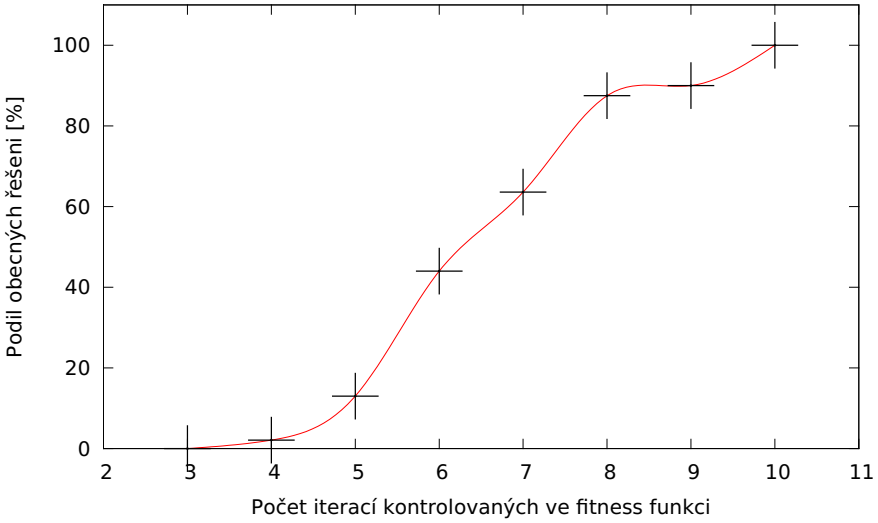


Obrázek 7.4: Závislost počtu nalezených řešení na počtu iterací kontrolovaných ve fitness funkci. Vodorovná osa udává, kolik členů posloupnosti muselo být nalezeno.

Jak je z tabulky patrné, nemá počet iterací příliš velký vliv na průměrný počet evaluací. Výjimkou je snad jen řádek se třemi iteracemi, ve kterém je tento počet znatelně nižší. Pokud si ovšem uvědomíme, že nalezení řešení v tomto případě spočívá pouze ve vygenerování sekvence 1, 2, 6, je zřejmé, že řešení bude nalezeno rychle. Ani jedno z nalezených řešení však není obecné. V tomto případě jsou se rozhodl prezentovat výsledky kromě tabulky také ve formě grafů 7.4 a 7.5. V obou těchto grafech jsou výsledné body proloženy křivkou, která přibližně znázorňuje trend závislosti nalezených řešení na počtu iterací kontrolovaných ve fitness funkci.

V grafu 7.4 je jasně vidět, že počet nalezených řešení se s rostoucím počtem kontrolovaných iterací rapidně snižuje. To je pochopitelné, protože s rostoucím počtem iterací se v tomto případě zvyšuje i složitost chromozomu, který takovou funkci generuje a je tedy obtížnější jej nalézt.

Oproti tomu graf 7.5 znázorňuje, jakou měrou se zvyšuje podíl obecných řešení. To je opět logické, protože nekvalitní řešení neprojdou již přes předchozí iterace. U 10. iterace jsou obecná dokonce všechna nalezená řešení.



Obrázek 7.5: Podíl obecných řešení z celkového počtu nalezených řešení. Vodorovná osa udává počet prvních členů posloupnosti, které byly kontrolovány ve fitness funkci.

7.3.2 Bez využití násobení

Cílem této části bylo zjistit, zda je funkci faktoriál možné implementovat i bez využití násobení, jako tomu bylo u druhých mocnin. Z dostupných funkcí tedy odebereme funkci DMULT. Z parametrů evoluce se změní pouze následující:

```
COL_COUNT                    20
```

Tato změna byla provedena, protože lze očekávat, že bez použití násobení bude k nalezení řešení nutný rozsáhlejší prostor. Vyhodnocení probíhá stejně jako v předešlém případě, ale maximální počet evaluací je nastaven na 10 000 000.

Po provedení 100 běhů, což bylo vzhledem k nastavenému maximálnímu počtu evaluací časově velmi náročné, jsem prošel výsledky a zjistil jsem, že řešení bylo nalezeno pouze v 1 případě a nebylo obecné. Na druhou stranu jsem po prohlédnutí prvních výsledků nečekal, že by řešení bylo vůbec nalezeno, neboť většina chromozomů dokázala po 10 000 000 evaluací generovat maximálně pět prvků posloupnosti. Tento výsledek tedy lze považovat do jisté míry za úspěch. Nicméně můžeme konstatovat, že faktoriál realizovaný pouze pomocí sčítání není pro SMCGP nejvhodnější funkce.

Tabulka 7.7: Shrnutí výsledků hledání posloupnosti mocnin

Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
424 148	3 833	45	84,4%

7.4 Posloupnost mocnin

Cílem tohoto experimentu je dokázat, že je pomocí SMCGP možné generovat posloupnost mocnin ve tvaru $x + x^2 + x^3 + \ldots$. U tohoto experimentu jsem použil základní funkce včetně násobení a všechny sebemodifikující funkce. Fitness hodnota se počítá následujícím způsobem:

Nechť $\vec{w} = (w_1, w_2, \ldots, w_n)$ je požadovaná posloupnost a $\vec{o} = (o_1, o_2, \ldots, o_n)$ posloupnost generovaná programem. Fitness funkce je potom součtem všech rozdílů výstupního a chtěného podílu po sobě následujících hodnot, tedy:

$$fitness(\vec{o}) = \sum_{i=1}^{n-1} |\frac{o_{i+1}}{o_i} - \frac{w_{i+1}}{w_i}|$$

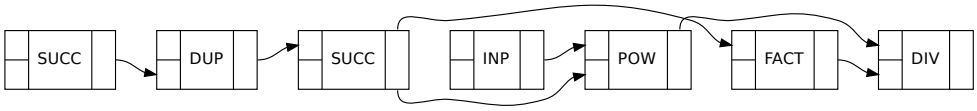
Tím je zajištěna příznivější hodnota fitness i u posloupností ve tvaru $kx+kx^2+kx^3+\ldots$. Maximální počet evaluací je omezen na 1 000 000. Je to poměrně nízká hodnota, ale vzhledem k tomu, že se jedná o triviální funkci, tak je postačující. V tomto experimentu nejde o to nalézt nejlepší řešení, ale dokázat, že řešení je možné nalézt, abychom jej mohli uvažovat v následujícím experimentu. Provede se opět 100 běhů programu. Parametry evoluce jsou následující:

```
GENE_IN_COUNT           2
GENE_OUT_COUNT          1
CHROMOSOME_IN_COUNT     1
CHROMOSOME_OUT_COUNT    1
VAR_TYPE                double
COL_COUNT               15
ROW_COUNT               1
L_BACK                  5
LAMBDA                  4
MUTATE_MAX              3
MAX_SELFMOD             2
```

Výsledky evoluce shrnuté v tabulce 7.7 sice nejsou nejlepší, ale jak již bylo zmíněno v úvodu k tomuto experimentu, cílem bylo dokázat, že lze najít takové obecné řešení. Cíl byl tedy splněn a experiment můžeme prohlásit za dokončený.

Tabulka 7.8: Shrnutí výsledků hledání členů Taylorova rozvoje

Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
50	50	100	100%



Obrázek 7.6: Chromozom využívající funkcí POW a FACT, který generuje Taylorovu posloupnost libovolné délky

7.5 Taylorův rozvoj

Předchozí dva experimenty směřovaly k realizaci většího experimentu, jímž je evoluce chromozomu generujícího členy Taylorovy řady. Většina Taylorových řad obsahuje pouze mocniny vstupní proměnné a faktoriál. Pro začátek zkusíme implementovat Taylorovu řadu tak, že přidáme dvě nové funkce uzlu POW a FACT. Konkrétně budeme implementovat řadu

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

Vyhodnocení fitness funkce probíhá podobně jako v předchozích případech tak, že fitness funkce udává počet správně vygenerovaných prvků. Maximální počet evaluací je omezen na 5 000 000. Řešení musí správně generovat prvních 10 členů posloupnosti. Poté se pomocí dalších následujících prvků zkontroluje, zda funkce zobecňuje. Parametry evoluce jsou následující:

```
GENE_IN_COUNT              2
GENE_OUT_COUNT             1
CHROMOSOME_IN_COUNT        1
CHROMOSOME_OUT_COUNT       1
VAR_TYPE                   double
COL_COUNT                  20
ROW_COUNT                  1
L_BACK                     7
LAMBDA                     4
MUTATE_MAX                 2
MAX_SELFMOD                2
```

Bylo provedeno 100 běhů, jejichž sumarizované výsledky jsou zapsány v tabulce 7.8.

Výsledky jsou dle očekávání velmi příznivé. Pokud jsou funkce pro výpočet mocniny vstupu a faktoriálu implementovány programově, nalezne se řešení již při generování počátečních 50 chromozomů, neboť stačí tyto funkce vhodně propojit se vstupy a blokem realizujícím dělení. Na obrázku 7.6 je zobrazeno jedno z nalezených řešení. Tento chromozom dokáže generovat Taylorovu řadu o libovolné délce.

Tabulka 7.9: Shrnutí výsledků hledání Taylorova rozvoje s použitím funkce POW

Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
5 208 990	1 152 716	10	60%

Tabulka 7.10: Shrnutí výsledků hledání Taylorova rozvoje s použitím funkce FACT

Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
5 785 314	321 761	10	2%

Zajímavějším experimentem bude nepochybně implementace stejné Taylorovy řady, tentokrát však již bez použití funkce FACT. Tento experiment je podstatně složitější, neboť funkce pro výpočet faktoriálu se musí vyvinout evolučně. Způsob vyhodnocování fitness funkce zůstává stejný, změní se však maximální povolený počet evaluací, konkrétně na hodnotu 10 000 000. Ostatní hodnoty zůstávají.

Po provedení 100 běhů však nebylo nalezeno žádné řešení. Při bližším zkoumání hodnot bylo zjištěno, že fitness hodnoty chromozomů jsou většinou rovny polovině, tedy nalezení pěti prvků posloupnosti. Vzhledem k tomu, že tato hodnota fitness byla stejná po několik milionů evaluací, nelze předpokládat, že by zvětšení maximálního povoleného počtu evaluací vedlo k lepšímu výsledku.

Zkoušel jsem experimentovat s různými hodnotami nastavení evoluce, což nevedlo k řešení. Potom jsem vyzkoušel několik různých fitness funkcí. Kromě již zmíněné funkce udávající počet správně určených prvků jsem zkoušel i fitness funkci vracející vzdálenost generovaných a požadovaných prvků. Dále jsem se pokusil nalézt řešení s fitness funkcí udávající podíl po sobě jdoucích hodnot. Ani jedno z těchto řešení však nevedlo k lepším výsledkům.

K vyhodnocení správnosti řešení byla u tohoto experimentu použita poměrně vysoká hodnota počtu iterací. Jak bylo popsáno u faktoriálu, je použití vysoké hodnoty problematické. Všechna nalezená řešení sice pravděpodobně budou obecná, ale šance nalézt takové řešení bude poměrně nízká. Naproti tomu, když se bude kontrolovat pouze pět prvků místo deseti, bude šance na nalezení řešení vyšší a poměr obecných řešení je v tomto případě také poměrně vysoký. Výsledné hodnoty jsou uvedeny v tabulce 7.9.

Obdobným způsobem jsem postupoval v dalším případě, kdy byla naopak ponechána funkce FACT a odstraněna funkce POW. Výsledky této části jsou v tabulce 7.10.

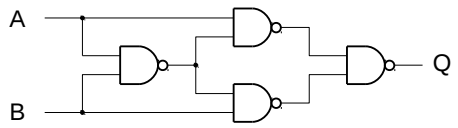
Na závěr tohoto experimentu zůstává vyzkoušet, zda je možné evolvovat program generující členy Taylorova rozvoje bez použití funkcí POW a FACT. Postupoval jsem opět stejně jako v předchozích případech, hodnoty zůstaly na počátku nastaveny stejně. Řešení bohužel nebylo nalezeno. Provedl jsem tedy mnoho úprav, mezi jinými i úpravu fitness funkce, aby zvýhodňovala chromozomy schopné generovat posloupnost mocnin (tedy čitatel požadované Taylorovy řady). Ani po těchto úpravách se však řešení nepodařilo nalézt. Výsledkem tohoto experimentu tedy je poznatek, že SMCGP není příliš vhodné, pokud je třeba, aby se samostatně vyvinulo více částí, které by se na závěr vhodným způsobem propojily. Je to způsobeno především tím, že se velmi špatně hledá fitness funkce, která by podporovala současnou evoluci více zamýšlených částí programu.

Tento problém jsem se pokusil řešit převodem chromozomu na dvojrozměrnou reprezentaci, přičemž cílem bylo, aby se mohla řešení vyvíjet nezávisle na sobě v různých řádcích. Narazil jsem však na problém týkající se sebemodifikace. Pokud je sebemodifikace implementována tak, že se pracuje s celými sloupci, aby byl počet uzlů chromozomu stále násobkem počtu řádků, může evoluce jednoho funkčního bloku negativně ovlivnit evoluci funkcí

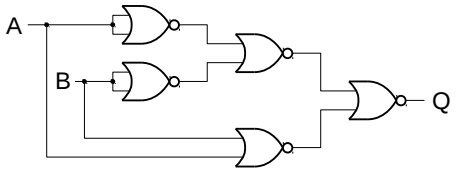
v ostatních řádcích. Aby bylo nalezeno řešení, musely by se funkce vygenerovat vhodně zarovnané, aby se při sebemodifikaci kopírovaly požadované uzly obou. Takové řešení sice existuje a ručně jsem ho vytvořil a otestoval, avšak prohledávaný prostor řešení je natolik rozsáhlý, že šance vygenerování dvou řádků, kdy každý plní jednu funkci a současně jsou tyto řádky vůči sobě vhodně zarovnané, je velmi malá.

Výše zmíněný problém by se dal řešit tak, že by evoluce jednotlivých funkcí probíhala v samostatných řádcích bez ohledu na ostatní řádky. Tento přístup by bylo možné implementovat. Uzly by se kopírovaly pouze v rámci jednoho řádku a ostatní řádky by se na konci doplňovaly novými bloky, aby bylo zachováno obdélníkové uspořádání. Jedná se však o tak výraznou úpravu, že jsem se ji rozhodl neimplementovat, neboť by dle mého názoru takové rozšíření již bylo mimo rozsah této práce.

Na závěr lze tedy konstatovat, že SMCGP není v současné podobě na Taylorovy řady příliš vhodné.



Obrázek 7.7: Realizace hradla XOR pomocí hradel NAND



Obrázek 7.8: Realizace hradla XOR pomocí hradel NOR

7.6 Parita

Tento experiment má za cíl vyvinout chromozom, který bude realizovat obvod pro výpočet parity. Vyvinutý program by měl být obecný, což znamená, že v každé iteraci bude generovat obvod pro výpočet parity, který má o 1 vstup více. Dále je určeno, že obvod smí využívat pouze hradla typu AND, OR, NAND, NOR. Základním stavebním prvkem pro výpočet parity je obvykle hradlo typu XOR. Naše funkční sada tuto funkci neobsahuje, ovšem je možné ji z obsažených funkcí vytvořit. Postup vytvoření hradla XOR z hradel typu NAND a NOR je zakreslen na obrázcích 7.7 a 7.8. Díky tomu by mělo být možné obvod pro výpočet parity pomocí těchto funkcí realizovat.

Jednou z nejdůležitějších částí každého evolučního algoritmu je vhodně navržená fitness funkce. Je tomu tak i v tomto případě. Fitness se počítá jako počet správně určených výsledků během iterací. Funkce je však upravena tak, že pokud obvod nenalezne zcela správné řešení pro danou iteraci, nebude se pokračovat k další iteraci. Tím je zajištěna nejen vyšší rychlost vyhodnocování, ale především to, že jedinec, který negeneruje správné řešení ani v počátečních iteracích bude mít nižší hodnotu fitness, než jedinci, kteří v průběhu počátečních iterací generují správné řešení, ale v pozdějších iteracích již nikoliv. Pokud chceme nalézt obecné řešení, je tento postup více než žádoucí.

Pro zrychlení výpočtu jsem provedl ještě jedno opatření. Při počítání fitness funkce se nekontrolují všechny dostupné kombinace vstupních hodnot, ale používá se náhodně vygenerovaná množina trénovacích vektorů. Velikost tohoto vektoru pochopitelně závisí na počtu výstupů kontrolovaných ve fitness funkci. V tomto případě se kontroluje například 9 iterací. Existuje tedy $2^9 = 512$ možných kombinací vstupních hodnot. Při paralelní simulaci je k vyzkoušení těchto kombinací třeba $\frac{512}{32} = 16$ proměnných o velikosti 32 bitů. Upravená fitness funkce místo toho využívá pouze 4 náhodné 32 bitové vstupy. Vyhodnocení fitness funkce by tedy mělo být až 4krát rychlejší. Tento přístup samozřejmě může produkovat i řešení, která nejsou obecná a jsou přizpůsobena pouze pro konkrétní vstupní vektory. Při ověření generalizace se však řešení vyhodnocuje důkladněji a tak se tato řešení elimi-

Tabulka 7.11: Srovnání počtu evaluací nutných k nalezení řešení při použití různých hodnot parametru L-back

L-back	Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
4	955 390	2 534	85	71,8%
6	1 834 860	45 918	84	69,3%

nují. Z výsledků bylo navíc zjištěno, že tento přístup generuje téměř stejný podíl obecných řešení. Příčinou může být to, že fenotypy v jednotlivých iteracích vznikají duplikováním uzlů fenotypu z předchozí iterace. Kopírují se tedy správně fungující funkční bloky z předchozích iterací, tím pádem je pravděpodobnost nalezení řešení přizpůsobeného pouze určitým vstupům poměrně nízká.

Generované chromozomy se vyhodnocují během 2. až 10. iterace. Pokud je nalezeno řešení, které v tomto rozsahu generuje správné výstupní hodnoty, ověří se funkčnost nalezeného řešení na dalších iteracích až po 20. Řešení, které generuje správné výsledky pro všechny iterace je považováno za obecné. Jedná se samozřejmě o zjednodušení, neboť tento postup nezajišťuje, že dané řešení bude generovat správné výsledky v iteracích vyšších než 20.

U tohoto experimentu je poměrně obtížné i nalezení nejvhodnějších parametrů evoluce, neboť při jejich změnách se výsledky značně liší. Především se mění průměrný počet evaluací vedoucí k nalezení řešení. Příkladem takové změny je například značný nárůst počtu evolucí potřebných k nalezení řešení v případě změny parametr `L_BACK` uvedený v tabulce 7.11.

Tento jev si vysvětluji tak, že řešení bude pravděpodobně obsahovat hradla XOR vytvořená z poskytnutých funkcí. K sestavení takového hradla je tedy třeba ponechat dostatečný počet uzlů. Pokud bude uzlů málo, může se stát, že bude v prohledávaném prostoru velmi málo konfigurací plnících funkci hradla XOR. Hledání takové konfigurace potom připomíná spíše hledání jehly v kupce sena, než evoluční návrh. Dále uvažme, že evolučně vytvořené hradlo XOR se musí v jednotlivých iteracích patřičným způsobem duplikovat. Vzhledem k tomu, že mezní indexy duplikačních funkcí se určují podle parametrů uzlu, které se mění pouze v určitém procentu mutací, je výpočetně náročnější nalézt evolucí konfiguraci, která duplikuje větší blok uzlů o pevně daném začátku a konci. Tento argument tedy zase stojí proti zbytečně velkému počátečnímu chromozomu, ve kterém by mohlo mít vyvinuté hradlo XOR zbytečně velkou délku a špatně by se kopírovalo. Experimentováním jsem nakonec došel k těmto parametrům evoluce, které poskytovaly nejlepší výsledky:

```
GENE_IN_COUNT           2
GENE_OUT_COUNT          1
CHROMOSOME_IN_COUNT     20
CHROMOSOME_OUT_COUNT    1
VAR_TYPE                unsigned long long
COL_COUNT               20
ROW_COUNT               1
L_BACK                  4
LAMBDA                  4
MUTATE_MAX              2
MAX_SELFMOD             2
```

Maximální počet vyhodnocení byl omezen na hodnotu 5 000 000. Po provedení 100

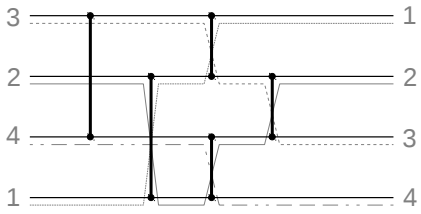
Tabulka 7.12: Shrnutí výsledků hledání obecného řešení parity

Prům. poč. vyh.	Min. poč. vyh.	Počet řešení	Obecná řešení
955 390	2 534	85	71,8%

běhů jsem došel k výsledkům, které jsou shrnuty v tabulce 7.12. Výsledky jsou poměrně uspokojivé. V porovnání s [6] je průměrný počet evaluací nutných k nalezení řešení vyšší. To může být způsobeno například jinak zvolenými parametry evoluce, neboť jak bylo zmíněno výše, mají i malé změny těchto parametrů značný vliv na výsledky, takže navržené hodnoty velmi pravděpodobně nebudou optimální.

Jinak jsou však výsledky experimentu velmi uspokojivé. V porovnání s ostatními evolučními technikami (viz tabulka 4.4) poskytuje SMCGP velmi dobrá řešení ve velmi nízkém počtu evaluací. Především jsou nalezena i obecná řešení, díky čemuž nelze výsledky v některých případech srovnat vůbec. Uvažme, že v tomto experimentu byla nalezena řešení, která počítají paritu o 20 vstupech. Skutečnou obecnost řešení by bylo možné ručně dokázat různými postupy, například postupem uvedeným v článku [6].

Generovat takto rozsáhlé obvody pomocí CGP je prakticky nereálné. Kromě toho nelze pomocí CGP nalézt obecné řešení. Výhody SMCGP oproti tradičnímu CGP jsou tedy v tomto případě zřejmé.



Obrázek 7.9: Řadicí síť se 4 vstupy. Šedou barvou je zobrazeno seřazení konkrétní posloupnosti.

7.7 Řadicí sítě

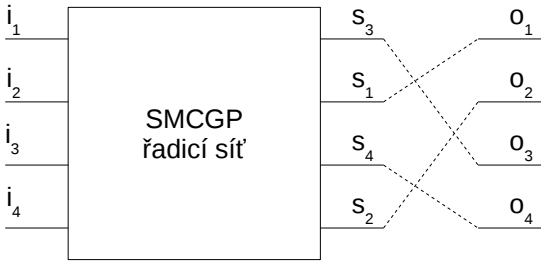
V tomto experimentu je cílem zjistit, zda SMCGP poskytuje lepší výsledky než CGP při návrhu řadicích sítí. Dále ověříme, zda zavedená rozšíření pomáhají nalezení řešení. Jedná se o rozšíření počtu výstupů uzlů a možnost pracovat v SMCGP s dvojrozměrným polem uzlů.

Řadicí síť je matematický model skládající se z vodičů a komparátorů, který lze využít k řazení posloupnosti čísel. Komparátor je vždy připojen ke dvěma vodičům. Funguje tak, že hodnoty nesené vodiči porovná, nižší hodnotu umístí na horní vodič, vyšší hodnotu na spodní. Při vhodném uspořádání komparátorů vznikne řadicí síť, na jejímž výstupu je seřazená vstupní posloupnost. Příklad řadicí sítě je znázorněn na obrázku 7.9. I přes jednoduchost modelu je problematika řadicích sítí velmi rozsáhlá a komplikovaná.

Při evolučním návrhu řadicích sítí lze uplatnit dva různé přístupy. Prvním přístupem je návrh řadicích sítí pro pevně daný počet vstupů. V tom případě je cílem evoluce většinou nalezení optimálního řešení. Optimální řadicí síť je taková, která k seřazení posloupnosti o určité velikosti potřebuje nejmenší možný počet komparátorů nebo dosahuje nejmenšího možného zpoždění. Pro nižší počty vstupů jsou optimální sítě známé a jejich optimálnost je dokázána. U vyšších počtů vstupů však nejsou optimální řešení známá a postupně se vyvíjejí stále lepší řadicí sítě. Například síť na obrázku 7.9 je, co se týče velikosti, optimální.

Druhým přístupem je evoluční návrh libovolně velkých řadicích sítí. V tomto případě evoluce nalezne řešení, které dokáže generovat řadicí síť pro libovolný počet vstupů. Je ovšem nutno podotknout, že tyto řadicí sítě nejsou alespoň u větších počtů vstupů optimální. Příkladem evolučního návrhu řadicích sítí je například návrh řadicích sítí pomocí developmentu uvedený v práci [2].

Důležitým prvkem je zde opět fitness funkce. Její vyhodnocení udává počet správně seřazených prvků. Problémem však v tomto případě byl způsob vyhodnocování chromozomu v SMCGP. Výstupem chromozomu je patřičný počet nejlevějších uzlů, které mají parametr isOutput nastavený na hodnotu true. To však v důsledku znamená, že i výstup správného řešení nemusí být ve správném pořadí. Tato situace je ilustrována na obrázku 7.10. Cílem řadicí sítě je porovnat vzájemně prvky. To je ve správném řešení zajištěno, neboť výstup SMCGP označený na obrázku s_1 skutečně obsahuje nejmenší prvek, výstup označený s_2 druhý nejmenší atd. Není však již zajištěno, že se uzly vyhodnotí v pořadí odpovídajícím tradiční řadicí síti, tedy od nejmenšího prvku k největšímu. Upravil jsem tedy fitness funkci tak, aby na pořadí výstupů nezáleželo. Zásadní je, aby při přivedení různých kombinací



Obrázek 7.10: Znázornění nesprávného pořadí výstupu seřazených prvků a řešení tohoto problému pomocí propojení výstupů SMCGP s odpovídajícími výstupy tak, aby bylo pořadí stejné jako u běžné řadicí sítě.

na vstupy byl vždy například nejmenší prvek na stejné pozici. Na obrázku je například nejmenší z řazených čísel vždy na pozici s_1, tedy na druhém výstupu. Jestli bude tato pozice první, druhá nebo poslední není důležité, protože na správné výstupy lze uzly na závěr připojit propojením příslušného uzlu s požadovanou pozicí. Toto propojení je na obrázku znázorněno čárkovaně. To by samozřejmě dokázala provést i evoluce pomocí uzlů typu NOP, ovšem zbytečně by se tím zvýšil počet evaluací nutný k nalezení řešení.

Dalším zásadním rysem fitness funkce je to, že se neporovnávají celá nebo reálná čísla, ale pouze binární hodnoty. To je možné díky tzv. principu nula-jedna. Je dokázáno (viz [8]), že pokud síť dokáže správně seřadit všechny možné vstupní kombinace složené pouze z binárních hodnot 0 a 1, je tato síť schopná řadit i hodnoty jiného typu. Rozdílem v realizaci těchto dvou přístupů je to, že při porovnávání čísel se využívají funkce MIN a MAX, zatímco při porovnávání binárních hodnot jsou to funkce AND a OR. Hlavní přínos tohoto poznatku tkví v tom, že můžeme provádět paralelní simulaci, tedy použít jako vstup např. 64 bitová čísla a vyhodnotit tak při jednom vyhodnocení sítě 64 různých kombinací vstupů. Tím je výpočet fitness funkce a tím pochopitelně i celého programu značně urychlen.

Kvůli možnosti paralelní simulace tedy bude základním stavebním prvkem evolvovaných řadicích sítí komparátor implementující vnitřně funkce AND a OR. Z těchto komparátorů je pak vytvořena celá síť. Nyní již můžeme postoupit k jednotlivým částem experimentu.

7.7.1 Evoluční návrh řadicí sítě o pevně daném počtu vstupů

V této části je cílem generovat řadicí síť o pevně daném počtu vstupů a ověřit, zda pro tento typ úlohy poskytují navržené modifikace algoritmu SMCGP zlepšení oproti výchozí variantě.

Sebemodifikace je v následující části experimentu uplatněna tak, že se sebemodifikace provede v každém kroku provede, vyhodnotí se a pokud je výsledek lepší, než nejlepší dosud nalezený, nahradí původního nejlepšího jedince. Nevýhodou tohoto přístupu je, že se jedinec může v průběhu generací neúměrně zvětšovat. Tomu lze zamezit různými způsoby, například zavedením horní hranice velikosti jedince.

Nejprve ověříme, jestli zavedení možnosti práce s dvojrozměrným chromozomem v SMCGP pomáhá v nalezení řešení. K tomu účelu zkusíme nalézt řadicí sítě o 3 a 4 vstupech.

Tabulka 7.13: Shrnutí výsledků hledání řadicí sítě o 3 vstupech

	Prům. poč. vyh.	Min. poč. vyh.	Počet řešení
Jednorozměrný chromozom	118 846	3 874	100
Dvojrozměrný chromozom	90 693	5 111	100

Tabulka 7.14: Shrnutí výsledků hledání řadicí sítě o 4 vstupech

	Prům. poč. vyh.	Min. poč. vyh.	Počet řešení
Jednorozměrný chromozom	2 583 360	777 180	100
Dvojrozměrný chromozom	4 986 245	955 410	100

Aby bylo možné výsledky jednorozměrných a dvojrozměrných řešení srovnat, je nutné, aby měly generované chromozomy stejné parametry. Především se jedná o to, aby byl počet genů v obou případech stejný. Počet sloupců jednorozměrné varianty tedy bude odpovídat součinu počtu řádků a sloupců dvojrozměrné varianty. Obdobně parametr L-back bude u jednorozměrné sítě roven parametru L-back dvojrozměrné sítě vynásobenému počtem řádků.

Parametry evoluce jsou tedy následující:

```
GENE_IN_COUNT              2
GENE_OUT_COUNT             1
CHROMOSOME_IN_COUNT        3|4
CHROMOSOME_OUT_COUNT       3|4
VAR_TYPE                   unsigned long long
COL_COUNT                  6|15/18|60
ROW_COUNT                  3|4/1
L_BACK                     3|4/9|16
LAMBDA                     4
MUTATE_MAX                 2
MAX_SELFMOD                2
```

Hodnoty uvedené ve složených závorkách oddělené svislou čarou značí hodnoty příslušných parametrů pro síť o 3 a 4 vstupech. Lomítkem jsou odděleny parametry dvojrozměrné a jednorozměrné varianty.

Provedl jsem 100 běhů a jejich výsledky jsou v tabulkách 7.13 a 7.14.

Jak je z výsledků jasně patrné, nemá zavedení druhého rozměru pro nalezení řešení příliš velký význam. Pokud bychom netrvali na stejném počtu genů a příslušně upraveného parametru L-back, dostali bychom pomocí jednorozměrného řešení ještě lepší výsledky, které jsou uvedeny v tabulce 7.15.

Nyní je již zřejmé, že pro tento experiment nenabízí dvojrozměrná reprezentace jedince

Tabulka 7.15: Shrnutí výsledků hledání řadicí sítě o 3 a 4 vstupech pomocí jednorozměrného chromozomu

	Prům. poč. vyh.	Min. poč. vyh.	Počet řešení
3 vstupy	7 744	274	100
4 vstupy	535 636	33 444	100

Tabulka 7.16: Shrnutí výsledků pro různý počet výstupů genů

	Prům. poč. vyh.	Min. poč. vyh.	Počet řešení
1 výstup	15 632	479	100
2 výstupy	7 744	274	100

žádné znatelné výhody. Osobně však věřím, že v určitých typech úloh má dvojrozměrná reprezentace opodstatnění.

Další částí experimentu je ověření, zda při řešení tohoto problému poskytuje nějakou výhodu zavedení více výstupů. Teoreticky by zde měly být uzly se dvěma výstupy realizující komparátor výhodnější, než uzly s jedním výstupem realizující samostatně funkce MIN a MAX (resp. AND a OR).

Opět bylo provedeno 100 běhů, přičemž se hledala 3vstupá řadicí síť. Výsledky jsou shrnuty v tabulce 7.16.

V tomto případě je přínos rozšíření modelu naprosto zřejmý. Potvrdil se tedy předpoklad, že pro řadicí sítě je výhodnější použít geny se dvěma výstupy. Dále předpokládám, že toto rozšíření by nalezlo využití v celé řadě dalších úloh.

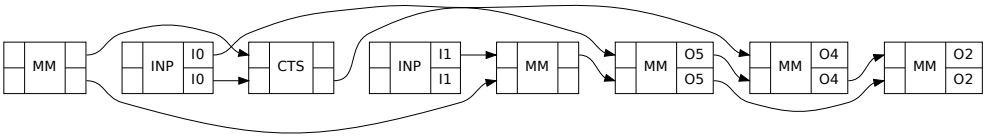
Z výše uvedených výsledků jasně vyplývá, že SMCGP není vhodnou technikou pro evoluci řadicích sítí o pevně daném počtu vstupů. Podařilo se evolvovat sítě maximálně o 6 vstupech. SMCGP je tedy v porovnání horší než CGP, což jsem nepředpokládal. Zkusil jsem tedy provést několik experimentů, kdy jsem zcela vyřadil sebemodifikující funkce. Výsledky však nebyly lepší. Předpokládám tedy, že neúspěch SMCGP při evoluci řadicích sítí netkví v sebemodifikujících funkcích, ale v jiné změně oproti CGP. Podle mého názoru je to způsobeno změnou práce se vstupy a výstupy. V tradičním CGP se může libovolný uzel připojit na libovolný vstup. V SMCGP se uzel musí připojit na uzel INP, který je v rozmezí daném parametrem L-back. Příslušný uzel INP však navíc nabízí pouze jednu hodnotu vstupu. Tím se značně snižují možnosti připojení ke vstupům, což může značně zhoršit schopnost nalézt řešení. Problémy s vyhodnocováním výstupů byly diskutovány výše. Tuto část bych tedy uzavřel tak, že se prokázala účinnost modifikace umožňující používat geny s více výstupy, ale SMCGP není vhodnou technikou pro evoluci řadicích sítí o pevně daném počtu vstupů.

7.7.2 Evoluční návrh řadicích sítí s libovolným počtem vstupů

Cílem této části je ověřit, zda je pomocí SMCGP možné evolvovat řadicí sítě o libovolném počtu vstupů. Hledaným řešením je genotyp, z nějž se v každé iteraci generuje fenotyp reprezentující řadicí síť schopnou řadit vždy o 1 prvek více než síť z předchozí iterace. Způsob vyhodnocení fitness funkce zůstává stejný jako v předchozí části.

Prvním pokusem bylo hledání řešení pomocí původní varianty SMCGP. V tomto případě však evoluce nalezla pouze řešení schopné řadit maximálně 5 vstupů. V dalších iteracích již řazení nebylo úspěšné. Podrobné prozkoumání výsledků odhalilo, že během evoluce zmizely z genotypu sebemodifikující funkce a fenotypy byly během iterací zcela stejné jako genotyp. Nalezeným řešením tedy byla síť, která dokázala řadit od 2 do 5 vstupů pouze v závislosti na počtu dodaných vstupů.

Další provedený experiment tedy měl za cíl vynutit použití sebemodifikace tím, že počáteční genotyp obsahoval méně uzlů, než bylo nutné pro nalezení správného řešení. Konkrétně se jednalo o hledání řadicí sítě, která dokáže řadit až 4 vstupy. Minimální počet komparátoru pro síť řadicí 4 vstupy je 5. Dále je nutné, aby genotyp obsahoval 4 uzly typu INP



Obrázek 7.11: Řešení umožňující z genotypu o velikosti 8 uzlů generovat řadicí sítě pro 2 až 4 vstupy.

pro načtení vstupů. Minimální počet uzlů nutných k nalezení řešení je tedy 9. Velikost genotypu byla nastavena na 8 sloupců, tedy hodnotu nižší, aby byla evoluce nucena použít sebemodifikující funkce. Během experimentu bylo skutečně nalezeno řešení, které dokáže z genotypu o velikosti 8 uzlů v průběhu iterací vytvářet řadicí sítě pro 2 až 4 vstupy. Jedno z nalezených řešení je zobrazeno na obrázku 7.11.

Vzhledem k získaným výsledkům a provedené analýze byla přidána nová sebemodifikující funkce EXP. Tato funkce je téměř stejná jako funkce DUP s tím rozdílem, že funkce EXP v bodě vložení duplikovaný uzlů odstraní jeden uzel. K zavedení této funkce vedl především fakt, že při pokusu o ruční vytvoření řadicích sítí pro 2, 3 a 4 vstupy jsem zjistil, že je nutné, aby se první vstupní uzel nejlevějšího komparátoru nahradil určitou sekvencí uzlů. Totéž platí pro nejpravější uzel, avšak sekvence vkládaných uzlů je odlišná.

Po provedení několika experimentů bylo zjištěno, že modifikovaná varianta algoritmu SMCGP dokáže provádět příslušná nahrazení, nedokáže ovšem takto přidané funkční bloky vhodným způsobem propojit, neboť jejich vzdálenost s iteracemi lineárně roste. Řešením by mohlo být zavedení nových typů uzlů umožňujících zápis/čtení paměti sdílené všemi uzly. Tímto způsobem by si mohly předávat hodnoty i uzly, které jsou od sebe více vzdálené. Problémem však zůstává způsob vyhodnocování fenotypu, neboť se postupuje od výstupních uzlů zpět ke vstupním a nemusí tedy být zaručeno, že by uzly prováděly zápis/čtení v požadovaném pořadí.

Tento experiment tedy prokázal, že algoritmus SMCGP není v základní ani modifikované variantě vhodný k evolučnímu návrhu řadicích sítí o libovolném počtu vstupů.

Kapitola 8

Závěr

Z předchozích kapitol je patrné, že sebemodifikující varianta kartézského genetického programování má nepochybně své opodstatnění a umožňuje řešit i problémy, které by klasickým kartézských genetickým programováním nebyly řešitelné. Z tohoto pohledu se mi tato oblast jeví jako velmi perspektivní. Kromě toho je tato varianta CGP natolik nová, že existuje ještě mnoho neprozkoumaných oblastí a možnosti pro samostatný výzkum jsou tedy značné.

Experimenty řešené v rámci práce ukázaly výhody a nevýhody SMCGP. Mezi hlavní výhody bezesporu patří možnost generovat obecná řešení. Další výhodou je možnost měnit chování programu za běhu, což může být v některých případech velmi užitečné. Nespornou výhodou je dle mého také zrychlení nalezení řešení u určitých typů problémů (např. výpočet parity). Nevýhodou SMCGP je například způsob práce se vstupy, který je vhodný pro generování obecných výsledků, avšak u experimentů zaměřených na nalezení jednoho konkrétního řešení je tento způsob zpracování vstupů spíše na škodu.

Myslím, že tato práce poměrně dobře ukázala, jaké úlohy jsou pro zpracování pomocí SMCGP nejvhodnější. Obecně se jedná o úlohy, které lze nějakým způsobem vyjádřit rekurzí. V takovém případě je možné ze základního chromozomu v průběhu iterací vyvíjet rozsáhlejší řešení. Typickým příkladem vhodného problému je Fibonacciho posloupnost.

Nevhodné jsou naopak problémy, které se rekurzivně vyjadřují špatně. Příkladem může být například návrh optimálních řadicích sítí. Optimální řešení řadicích sítí si totiž pro různý počet vstupů nejsou téměř podobné. Kvůli tomu nelze použít největší výhodu SMCGP, což je sebemodifikace. Navíc kvůli způsobu práce se vstupy, kdy není možné libovolný uzel připojit na libovolný vstup přímo trvá nalezení řešení i pro méně vstupů poměrně dlouho.

Zvláštním případem je například generování Taylorovy řady. Tuto úlohu lze bez problémů vyjádřit rekurzivně, avšak obecné řešení se nepodařilo nalézt. Bylo to dle mého způsobeno tím, že by bylo třeba vyvíjet současně funkci pro výpočet mocnin vstupu a funkci pro výpočet faktoriálu. Takový paralelní vývoj však není v této variantě SMCGP úspěšný. Řešením by mohlo být modifikovat SMCGP tak, aby bylo možné provádět více na sobě nezávislých evolucí, které by navzájem neovlivňovaly svoje geny. To by bylo možné implementovat například pomocí reprezentace jedince dvojrozměrným polem a možností provádět sebemodifikující funkce také v rámci řádků.

Navržená rozšíření se ukázala jako funkční, ovšem u dvojrozměrné reprezentace jedince nebyl zjištěn žádný přínos. Domnívám se, že jsem pravděpodobně špatně zvolil úlohu pro ověření tohoto konceptu a u jiných úloh by toto rozšíření mohlo vést ke zlepšení. Ostatní rozšíření se v určitých případech ukázala jako vhodná. Jedná se především o možnost pracovat s geny, které obsahují více než jeden výstup, což se například u řadicích sítí ukázalo jako výhodné. Tato změna tak dle mého svým způsobem přispěla i k zobecnění SMCGP.

Závěrem bych tedy svoji práci shrnul tak, že SMCGP je nepochybně perspektivní evoluční technikou, ovšem zatím jen pro určitý typ úloh, což ovšem platí obecně asi pro všechny evoluční techniky. Důležité tedy je volit pro daný problém správnou techniku, protože ani SMCGP nedokáže řešit vše.

Literatura

[1] Banzhaf, W.: Genotype-Phenotype-Mapping and Neutral Variation: A Case study in Genetic Programming. In *Davidor, Y., Schwefel, H.-P., R. Männer, R. (eds.): Proceedings of the conference on Parallel Problem Solving from Nature III.*, Springer-Verlag, 1994, s. 322–332.

[2] Bidlo, M.: *Evolutionary Design of Generic Structures Using Instruction-Based Development [dizertační práce].* Ústav počítačových systémů FIT VUT v Brně, 2009.

[3] Harding, S.; Miller, J.; Banzhaf, W.: Self-modifying cartesian genetic programming. In *Thierens, D., Beyer, H.-G., et al. (eds.) GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation, vol. 1*, ACM Press, 2007, s. 1021–1028.

[4] Harding, S.; Miller, J.; Banzhaf, W.: Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing. In *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, *LNCS*, ročník 5481, Tuebingen: Springer, 2009, s. 133–144.

[5] Harding, S.; Miller, J. F.; Banzhaf, W.: Evolution, development and learning using self-modifying cartesian genetic programming. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, Montreal: ACM, 2009, s. 699–706.

[6] Harding, S.; Miller, J. F.; Banzhaf, W.: Self Modifying Cartesian Genetic Programming: Parity. In *2009 IEEE Congress on Evolutionary Computation*, IEEE Computational Intelligence Society, Trondheim, Norway: IEEE Press, 2009, s. 285–292.

[7] Kampis, G.: Self-Modifying Systems In Biology And Cognitive Science: A New Framework for dynamics, information and complexity. 1991.

[8] Knuth, D. E.: *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973, ISBN 0-201-03803-X.

[9] Koza, J.: *Genetic programming: on the programming of computers by natural selection.* MIT Press, 1992.

[10] Langdon, W. B.; Poli, R.: Fitness Causes Bloat. Technická Zpráva CSRP-97-09, University of Birmingham, School of Computer Science, Birmingham, B15 2TT, UK, 1997.

[11] Miller, J.: What Bloat? Cartesian Genetic Programming on Boolean Problems. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, San Francisco, California, USA, 2001, s. 295–302.

[12] Miller, J.; Thomson, P.: Cartesian genetic programming. In *Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000*, Springer, 2000, s. 121–132.

[13] Miller, J. F.; Thomson, P.; Fogarty, T. C.: Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. 1997.

[14] Sekanina, L.: *Evolvable components : from theory to hardware implementations.* Springer-Verlag, 2004.

[15] Spector, L.; Stoffel, K.; Y, K. S.: Ontogenetic Programming. In *Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L. (eds.) Genetic Programming 1996: Proceedings of the First Annual Conference*, MIT Press, 1996, s. 394–399.

Příloha A

Obsah CD

Přiložené CD obsahuje tuto práci v elektronickém formátu pdf a zdrojové soubory pro sazbu práce pomocí programu LaTeX. Dále CD obsahuje všechny soubory potřebné k úspěšné kompilaci a spuštění programu. Konkrétní adresářová struktura CD je následující:

Adresář	Popis obsahu
\	Kořenový adresář
`dist`	Spustitelné soubory pro systém Linux
`src`	Zdrojové soubory k programové části práce
`factorial_add`	Zdrojové soubory experimentu Faktoriál bez využití násobení
`factorial_mult`	Zdrojové soubory experimentu Faktoriál s využitím násobení
`fibonacci`	Zdrojové soubory experimentu Fibonacciho posloupnost
`parity`	Zdrojové soubory experimentu Parita
`powers`	Zdrojové soubory experimentu Posloupnost mocnin
`sort_fixed`	Zdrojové soubory experimentu Řadicí sítě s pevně daným počtem vstupů
`sort_arbit`	Zdrojové soubory experimentu Řadicí sítě s libovolným počtem vstupů
`squares_add`	Zdrojové soubory experimentu Druhé mocniny bez využití násobení
`squares_mult`	Zdrojové soubory experimentu Druhé mocniny s využitím násobení
`taylor_both`	Zdrojové soubory experimentu Taylorova řada s využitím funkcí FACT a POW
`taylor_fact`	Zdrojové soubory experimentu Taylorova řada s využitím funkce FACT
`taylor_pow`	Zdrojové soubory experimentu Taylorova řada s využitím funkce POW
`tex`	Zdrojové soubory pro LaTeX
`fig`	Obrázky vyžadované šablonou práce
`img`	Vlastní obrázky použité v práci

Příloha B

Kompilace a používání programové části

V této příloze je popsán postup kompilace programové části této práce a používání výsledných programů. Dále je popsán postup zpracování výsledků generovaných pomocí programů. Především jsou rozebrány možnosti vizualizace chromozomů pomocí nástrojů CGPView a graphViz.

B.1 Kompilace programu

Všechny programy lze zkompilovat pomocí libovolného překladače jazyka C podporujícího normu C99. Ve výchozím nastavení se používá překladač `gcc`. V adresáři `src` je umístěn hlavní soubor `Makefile`, který volá soubory `Makefile` umístěné v adresářích příslušejících jednotlivým experimentům. Po spuštění překladu se provedou patřičné operace a výsledné spustitelné soubory se zkopírují do nově vytvořeného adresáře `\build`. Možné parametry příkazu `make` jsou uvedeny v tabulce B.1.

V adresáři `dist` jsou programy umístěny ve spustitelné podobě. Všechny tyto spustitelné soubory byly přeloženy na serveru `merlin.fit.vutbr.cz`.

B.2 Používání programů

Všechny programy jsou napsány jako konzolové aplikace bez grafického rozhraní. Spouští se bez parametrů, neboť všechny potřebné parametry jsou definovány u každého experimentu v hlavičkovém souboru `SMCGPDefs.h`.

Po spuštění se začne na standardní výstup vypisovat průběh evoluce. Výpis lze do jisté míry ovlivnit pomocí konstant v souboru `SMCGPDefs.h`. Pomocí konstanty `PRINT_EVERY` lze určit, po kolika generacích se má vypisovat aktuálně nejlepší fitness hodnota. Další konstantou ovlivňující výstup je `PRINT_SOLUTION_DOT`, která určuje, zda se má v případě nalezení řešení toto řešení vypsat ve tvaru vhodném pro zpracování pomocí programu graphViz. Konkrétní zpracování těchto výstupů je popsáno v následující části.

Výstup programu lze přesměrovat do souboru a následně jej požadovaným způsobem zpracovávat, například z něj získávat statistická data. K tomuto účelu je třeba ze souboru nejprve získat pouze požadované hodnotu. To lze provést pomocí různých programů (například `grep`. Všechny význačné řádky, tj. zda bylo nalezeno řešení, jestli je řešení obecné,

Tabulka B.1: Použitelné parametry programu `make`

Parametr	Popis funkce
`all`	Zkompiluje zdrojové soubory všech experimentů
`clean`	Odstraní všechny soubory vygenerované při překladu
`factorial_add`	Zkompiluje experiment Faktoriál bez využití násobení
`factorial_mult`	Zkompiluje experiment Faktoriál s využitím násobení
`fibonacci`	Zkompiluje experiment Fibonacciho posloupnost
`parity`	Zkompiluje experiment Parita
`powers`	Zkompiluje experiment Posloupnost mocnin
`sort_fixed`	Zkompiluje experiment Řadicí sítě s pevně daným počtem vstupů
`sort_arbit`	Zkompiluje experiment Řadicí sítě s libovolným počtem vstupů
`squares_add`	Zkompiluje experiment Druhé mocniny bez využití násobení
`squares_mult`	Zkompiluje experiment Druhé mocniny s využitím násobení
`taylor_both`	Zkompiluje experiment Taylorova řada s využitím funkcí FACT a POW
`taylor_fact`	Zkompiluje experiment Taylorova řada s využitím funkce FACT
`taylor_pow`	Zkompiluje experiment Taylorova řada s využitím funkce POW

kolik bylo třeba generací k jeho nalezení a další, jsou od sebe dostatečně odlišeny, aby bylo filtrování snazší.

B.3 Vizualizace řešení

Nalezená řešení je většinou nutné nějakým způsobem vizualizovat, protože výpis parametrů jednotlivých uzlů není příliš dobře pochopitelný. Pro usnadnění vizualizace jsou v programu implementovány funkce `LogCGPView` a `LogDot`. Pomocí těchto funkcí lze chromozom vypsat ve tvaru, který lze použít jako vstup programu CGPView, resp. graphViz. Popis těchto funkcí je obsažen v kapitole věnované implementaci. V této části bude popsáno, jak lze tyto výstupy převést do určitých grafických formátů.

B.3.1 Vizualizace pomocí programu CGPView

Program CGPView byl vyvinut na Fakultě informačních technologií VUT v Brně. Jeho primárním účelem je grafické znázornění chromozomů tradičního CGP. Funkce `LogCGPView` tedy převádí chromozomy SMCGP do tvaru použitelného jako vstup programu CGPView. Převod se týká především vstupů, které jsou v SMCGP realizovány odlišným způsobem. Další odlišností modifikované varianty SMCGP oproti CGP je možnost používat uzly s více výstupy. V tom případě se každý uzel rozdělí na tolik samostatných uzlů, kolik je počet jeho výstupů.

Nyní se zaměříme na konkrétní zpracování. Výstupem funkce `LogCGPView` je chromozom zapsaný v následujícím tvaru:

```
{20,1, 20,1, 2,4,0}([20]0,0,20)([21]20,20,13)([22]0,20,7)([23]1,0,0)
([24]22,22,9)([25]0,0,0)([26]25,23,3)([27]24,23,5)([28]27,27,3)([29]28,25,3)
([30]26,27,2)([31]29,29,5)([32]30,30,16)([33]32,29,5)([34]32,31,3)
([35]31,34,3)([36]35,34,4)([37]34,33,3)([38]36,35,2)([39]38,38,18)(37)
```

Tento kód lze zkopírovat přímo do programu CGPView a chromozom zobrazit. Výhodou tohoto programu je skutečnost, že lze jednotlivým funkcím uzlů přiřadit schematické značky. Například při návrhu kombinačních obvodů je tak možné zobrazit výsledek přímo pomocí hradel.

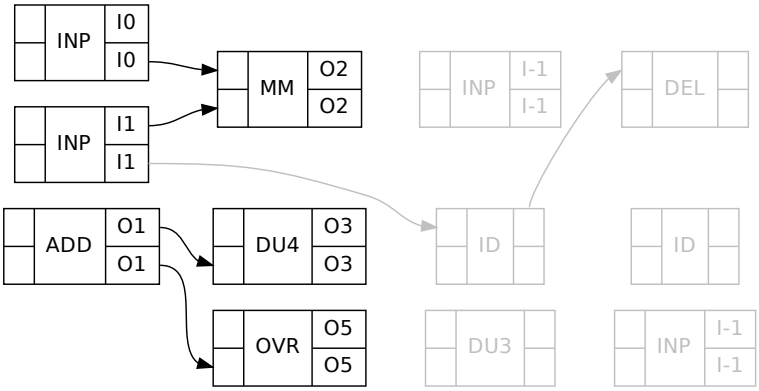
B.3.2 Vizualizace pomocí programu graphViz

Druhou možností je vizualizovat výsledné řešení pomocí programu graphViz. Funkce `LogDot` převede chromozom SMCGP na graf zapsaný pomocí jazyka DOT, který lze následně zpracovat. Hlavní výhodou tohoto přístupu je, že funkce `LogDot` je napsána tak, aby dokázala pracovat se všemi formami modifikované varianty SMCGP. Podporuje tedy i uzly s více výstupy, dvojrozměrná pole uzlů atd.

Zpracování výstupu této funkce probíhá tak, že vygenerovaný kód v jazyce DOT uložíme do samostatného souboru, například `chromozom.dot`. Kód chromozomu může mít například následující tvar:

```
digraph smcgp {
        rankdir=LR
        {rank = same;
        node0 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|INP|{<out0>I1|<out1>I1}}"]
        node1 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|ADD|{<out0>O1|<out1>O1}}"]
        node2 [color=black,fontcolor=black,shape=record,
                label="{{<in0>|<in1>}|INP|{<out0>I0|<out1>I0}}"]
        }
⋮
        edge [style=invis]
        node0->node3->node6->node9
        node1->node4->node7->node10
        node2->node5->node8->node11
        node0->node1->node2
        edge [style=solid]
        node2:out1->node3:in0 [color=black, weight=0]
        node0:out0->node3:in1 [color=black, weight=0]
        node1:out0->node4:in1 [color=black, weight=0]
        node1:out1->node5:in1 [color=black, weight=0]
        node0:out1->node7:in0 [color=gray, weight=0]
        node7:out0->node9:in0 [color=gray, weight=0]
}
```

Kód je z důvodu přehlednosti zkrácen. Celý kód tohoto chromozomu je vypsán v kapitole věnované implementaci. Všimněme si, že v první části jsou popsány jednotlivé uzly



Obrázek B.1: Znázornění chromozomu pomocí programu graphViz

včetně označení vstupů, výstupů a funkcí. Dále jsou barevně odlišeny aktivní a neaktivní uzly. Všechny uzly uzavřené v rámci jednoho bloku s příkazem `rank = same` jsou ve výsledném grafu umístěny pod sebou. To je vhodné pro zobrazení dvojrozměrných polí uzlů. V některých případech však může být výsledný graf názornější po odstranění tohoto příkazu. V takovém případě se totiž uzly rozmístí automaticky podle určitých pravidel tak, aby byl graf co nejčitelnější.

Ve druhé části jsou popsány propojení jednotlivých uzlů grafu. Všimněme si části mezi příkazy `edge [style=invis]` a `edge [style=solid]`. Jedná se o neviditelné hrany, které slouží pouze k zarovnání uzlů ve vertikálním směru. Program graphViz bohužel neumožňuje striktní zarovnání uzlů grafu a může se stát, že i při použití těchto hran nebudou uzly grafu správně zarovnané. Za těmito neviditelnými hranami již následují jednotlivá propojení uzlů, která jsou opět barevně značena tak, aby byly odlišeny aktivní a neaktivní uzly.

Zdrojový kód grafu v jazyce DOT lze pomocí programu graphViz exportovat do různých grafických formátů. Například export uvedeného kódu uloženého v souboru `chromozom.dot` lze exportovat do formátu pdf příkazem:

```
dot -Tpdf -o chromozom.pdf chromozom.dot
```

Výsledek by měl vypadat podobně jako na obrázku B.1. Exportovat lze pochopitelně do mnoha dalších formátů. Jejich úplný výčet lze nalézt v dokumentaci k programu graphViz.